学习JavaFX脚本语言(翻译)

作者: Ivan_Pig http://ivan-pig.javaeye.com

JavaFX

目 录

1. JavaFX

1.1 学习JavaFX脚本语言----1，2 3

1.2 学习JavaFX脚本语言----3,4 13

1.3 学习JavaFX脚本语言----5 22

1.4 学习JavaFX脚本语言----6，7 31

1.5 学习JavaFX脚本语言----8，9 49

1.6 学习JavaFX脚本语言----10，11（完） 62

1.1 学习JavaFX脚本语言----1，2

发表时间: 2008-12-06

很遗憾，JavaFX目前没有发布Linux版本！我使用
http://deadlock.netbeans.org/hudson/job/JavaFX_NB_daily/
的NetBeans日构建的插件(11月24日停止更新了)，运行JavaFX1.0的例子程序报错！郁闷，看来只能等Sun放出Linux版本的JavaFX了！还有，JavaFX目前的语法和之前的语法好像有了很大的区别！所以才想到翻译这个文档，计划一天翻译一到两节！
原文地址：http://java.sun.com/javafx/1/tutorials/core/index.html

Lesson 1: Getting Started with JavaFX Script

目录

- Step 1 : 下载并安装JDK
- Step 2 : 选择开发环境
- Step 3 : 下载并安装JavaFX编译器

Step 1: 下载并安装JDK

JavaFX脚本语言是建立在Java平台上的脚本语言，你的机器需要安装JDK5或者JDK6(JDK6在速度方面有提高)。如果你还没有安装，请在开始学习此教程前下载和安装JDK6或者JDK5。

Step 2: 选择开发环境

关于JavaFX开发环境，你有两个选择：使用IDE或者使用文本编辑工具。至于选择哪种开发环境，完全在于个人习惯。但是下面的总结能帮助你做出一个合理的选择。

普遍看法:（不翻译了，反正就是推荐使用IDE，也就是Netbeans了，目前就NetBean有此插件。要不你就用文本编辑工具吧！）

*IDEs present the most commonly used functions as Graphical User Interface (GUI) elements, and offer many useful features, such as automatic code completion. An IDE also gives you immediate feedback on errors and highlights code so that it is easier to understand.

* 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!)

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.

Step 3: 下载并安装JavaFX编译器

你还需要下载和安装JavaFX脚本编译器和运行时环境。一个途径就是下载JavaFX SDK，它除了提供JavaFX编译器，运行时环境还提供了一些其他的工具。

另一个途径就是从openjfx项目的网站上下载最新的编译器二进制文件。编译器是用Java写的，安装方法就是解压缩下载的文件，然后将javafxc和javafx tools添加到你的路径下去。具体的方法在PlanetJFX WiKi上有完整介绍！

最后，如果你想冒险尝试（if you want to live on the bleeding edge）----你可以加入OpenJFX编译器项目，创建你自己的一个拷贝，然后自己编译源码。（如果你选择这个途径，你还需要Apache Ant1.7.0,较新版本的Subversion---写此文章时最新版本是1.5.4）。想获得更多关于从源码创建编译器的信息，请浏览Planet JFX Wiki.

Lesson 2: Writing Scripts

目录

- 编写一个简单的计算器
- 声明变量
- 方法的定义和执行
- 给方法传递参数
- 方法的返回值
- 使用命令行参数

-编写一个简单的计算器

开始之前，你要有一个calculator.fx文件，包含如下代码。

```
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}");
}
```

JavaFX脚本语言是变异型语言，就是说所有的源代码在运行前都需要先转化为Java字节码！

下面的命令将会编译calculator脚本:

javafxc calculator.fx

编译完成后， 你会发现相应的Java字节码被生成，并且被放置到了叫做calculator.class的文件里面。你还会发现另一个文件calculator$Intf.class被创建。这个文件是被提供来运行应用的---你可以忽略它，但是不要删除。

现在你可以用下面的命令运行编译后的类了。

javafx calculator

输出：

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

这是一个很小的程序，但是它向你展现了JavaFX里面一些很重要的组成部分。学习这些部分是你掌握JavaFX的第一步。

Note:JavaFX语言不难掌握，但是由于你是第一次接触它，我们一次只介绍一部分的知识点以便你能更好的掌握它。我们的宗旨是方法的明了。我们会在后面的章节介绍更高级的用法。

-声明变量

让我们来仔细看一下calculator.fx这个例子---后面我们会扩展这个例子。

第一段代码定义了几个变量：

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

变量可以由var或者def关键字来定义。 两者的区别是var定义的变量可能在程序的执行过程期间有新的值被赋给它,而def定义的变量会一直保留第一次被赋予的值！我们给numOne,numTwo变量赋了值，但没有给result变量赋初值，因为这个变量将会保存我们的计算结果。

变量名一般由字母和数字组成，且不能以数字开头！建议以小写字幕开头，如果变量包含不止一个单词，从第二个单词开始，大写每个单词的第一个字母，如上例所示（numOne）。

Note:你可能注意到了，我们并没有做特别的规定让变量去存储一个数字（对字符串或其他类型的数据也是一样的）。编译器足够的聪明，能够从变量所存储的内容判断出类型。这就是类型猜测.类型猜测使你的工作简单化，像编写脚本语言一样，因为你不需要再定义变量的类型了。

-方法的定义和执行

余下的源代码定义了一些方法，add,subtract,multiply和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}");
}
```

方法是一段可执行的代码块，执行特定的任务！在我们的例子中，每个方法执行一个数学计算并打印结果！将执行的代码放置到方法中可以使你的程序更加的易读，易用和除错！方法体被大括号包围，方便辨认方法的开始和结束！

如果不调用方法，方法不会真正的执行！这使得你可以在你程序的任何地方去执行方法！不管方法的定义是在调用的前面还是后面，都没有关系！在我们的例子中，我们在方法定义前就执行了方法！

方法执行代码如下:

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

-给方法传递参数

下面我们将修改calculator代码使其能接受参数！参数是你在执行方法时传递给方法的值！使用这种途径，我们的计算器能运算任意两个数的四则运算，而不是只能运算硬编码给numOne和numTwo的值。

```
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}");
}
```

现在的输出为:

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

在这一个版本里面我们移除了numOne和numTwo这两个变量，因为不再需要它们了！取而代之的是我们修改了方法的定义，需要传递两个参数给方法。每个参数都有名称，后面跟着一个冒号加类型。当方法接受多个参数的时候，参数之间用逗号隔开。

-方法的返回值

方法可能会有返回值。比如，add方法可以修改为返回计算结果，如下：

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

}
```

add方法现在可以像这样运行：

```
var total;

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

如果没有返回值需要返回，默认返回Void。

-使用命令行参数

我们可以进一步的修改calculator程序来接受命令行参数。这可以使终端用户在运行时决定需要计算的数值。

```
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}");
}
```

这次修改新增了一些新的知识点，最值得注意的就是run()方法。不像其他的方法，run()是一个特殊的方法，是程序的入口点！run()方法会在args变量里面保存所有的命令行参数，以String Sequences的形式保存（Sequences是有序的对象链，很像其他编程语言里面的数组；在后面的章节将详细介绍）。

运行这段代码，现在你必须要设定第一个和第二个参数。

javafx calculator 100 50

输出:

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

Note:在之前的所有版本里面，我们并没有提供run()方法。我们只是输入代码然后它就执行了。默认情况下，编译器会插入一个无参的run()方法，然后在里面放入要执行的代码！

我们重新定义了numOne和numTwo变量，这次是才run()方法内定义的，我们的计算functions需要数字类型的参数，但是命令行参数是字符串，我们在将命令行参数赋给方法前，必须要要将每个命令行参数从String转化

为Integer。

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

我们借助Java语言来完成这个转换。这个简单的脚本语言能直接使用Java语言而获得很强大的功能（Tapping into the existing Java ecosystem as needed brings tremendous power to this otherwise simple scripting language.）

1.2 学习JavaFX脚本语言----3,4

发表时间: 2008-12-07

Lesson 3: Using Objects

目录

- 什么是对象？
- 声明一个对象
- 对象结构
- 执行实例方法

-什么是对象？

什么是对象？对象是软件里面互不关联的部分，对象具有状态和行为！简言之：

对象的变量表示对象的状态。
对象的方法表示对象的行为。.
理论上，对象能模拟任何东西，从GUI组件(按钮，多选框，标签)到不可见的抽象的东西(温度，金融，产品注册信息等)

Note:想了解更多，请看Java教程里的相应教程。

-声明一个对象

在JavaFX脚本语言里面，对象是由object literal创建的。

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

我们创建了一个Address对象，给假象的地址簿应用程序使用。下载Address.zip解压缩类文件和AddressBook.fx

Note:Address.class文件(Address.class和Address$Intf.class)包含了你创建Address对象的重要信息，供编译器使用。如果你想知道这些文件是从哪里来的，你可以先创建Address类的定义文件(叫Address.fx的文件)，然后编译它，就会生成Address.class文件。JavaFX脚本语言和Java语言提供了很多预编译的class文件供你在程序里面使用。这使得你可以完成不同领域的任务，包括创建有震撼视觉效果的GUI程序。我们会在此教程的最后告诉你怎么创建自己的类，在Writing Your Own Classes章节！这里，你只需要下载需要的类文件就可以了。

现在，编译脚本：javafxc AddressBoox.fx，如果没有任何的输出，则说明编译成功。

Note:技术术语，这里的变量应该叫做实例变量。你可以把实例变量想象成每个对象都会有的内建的属性。事实上，“属性”这个词是在以前版本里面使用的概念。在OOP的世界里面"instance"和"object"是同义的！

-对象结构

对象结构很容易学习和使用！第一个单词（Address）指出了你要创建的对象类型。两个大括号定义了对象的内容。对象里的每个实例变量都给了一个初始值。(street, city, state, zip)

多个对象可以在一起创建：

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

Address {
     street: "200 Pine Street";
     city: "San Francisco";
     state: "CA";
     zip: "94101";
}
```

Note:当定义一个object literal，实例变量可以以空格，逗号和分号隔开！下面的定义也是正确的：

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

Address {
     street: "200 Pine Street",
     city: "San Francisco",
     state: "CA",
     zip: "94101",
}
```

教程里面将使用分号来作为分隔符。当定义一个方法的时候，分号是必须的。
你也可以将一个刚创建的对象和一个之前创建的对象关联。

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

def addressTwo = Address {
     street: "200 Pine Street";
     city: "San Francisco";
     state: "CA";
     zip: "94101";
}
```

或者将一个对象放置到另一个对象里面！

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

在最新版本的例子里面，Customer定义了几个新的变量。而里面的address变量持有了一个Address对象。这种格式很平常，看看程序怎么缩进的。通过缩进，Address的变量和Customer里面的变量能很容易的辨认出来！要编译这个例子，下载Customer.zip,解压缩到相同的目录，编译即可。

-执行实例方法

JavaFX提供高了很多行为为你提供便利！这些行为是通过对象的方法提供的。

你通过变量的名字（这里是customer），后面跟个"."，紧接着是方法名称，这样的形式来执行实例方法。

```
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();
```

输出：

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

你现在可能想知道，这些方法是哪来的？我怎么知道一个对象里面包含那些变量和方法？如果你想使用一个类库，你需要API。API是一个格式良好的文档，列出了对象的变量和方法！这是唯一可以确定对象会提供什么方法的途径。在之后的教程里，在你要创建图形化程序的时候，你会学到如何的使用这个文档！

Lesson 4: Data Types
Contents

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

-String

你已经看过很多String的例子了，但是还是让我们来自习的看看它还具有那些特性。String的定义，可以用双引

号，也可以用单引号！

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

不论单引号还是双引号都必须是对称的：你能在双引号里面嵌入单引号，或者在单引号里面嵌入双引号。以单引号定义的String和双引号定义的String之间没有任何的区别！

你还能在String里面插入表达式，表达式以"{}"包围。

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

在表达式里面还能够再嵌入String：

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

在运行时，编译器会根据answer的值来使用"Yes"或者"No"自动的替换掉表达式。

要连接多个String，使用多个大括号即可。

```
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

Number 和Integer 接收数字类型的值，而很多情况下，你让编译器自己去猜测是什么类型就可以了。

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

当然，你可以指定变量的类型：

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

两者的区别是Number是浮点型而Integer是整型。只有当你需要浮点型的时候才使用Number，否则建议使用Integer.

-Boolean

Boolean 有两个值：true 和 false。当需要设置程序的特定状态时，使用此变量类型。

```
var isAsleep = true;
```

或者是一个条件表达式：

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

当()里面的值为true时，{}里面的代码将被执行。关于更多内容，请看Expressions lesson

-Duration

Duration 类型表示一系列的时间：

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

Durations被解释为时间---比如，5m就是5分钟。时间在animation 里将被频繁的使用。（请参看Building GUI Applications with JavaFX里面的Creating Animated Objects章节）

-Void

Void是被用来说明一个方法没有返回值的。

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

下面是等价的，省略了返回值

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

JavaFX里面的关键字Void，以大写V开头。如果你熟悉Java语言里面的void类型，请特别注意！

Note:在JavaFX里面，一切都是表达式。在第二个printMe里面返回值依然是Void，编译器能自动识别。在Expressions lesson你将会了解更多内容。

-Null

Null是一个特殊的值，表示一个变量没有一个正常值。Null不同于数字0或者空字符串，所以当Null和数字0或空字符串比较时，是不等的。

null关键字可以用来做比较，如下所示：

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

这个方法接收一个参数，判断是否为null.

1.3 学习JavaFX脚本语言----5

发表时间: 2008-12-09

Lesson 5: Sequences

目录

- 创建序列
- 使用布尔表达式创建序列
- 访问序列的元素
- 向序列里插入项
- 从序列里删除项
- 倒序序列里的项
- 比较序列
- 使用序列片段

-创建序列

除了5种最基本的数据类型。JavaFX脚本语言还提供了链式数据结构。序列表示一组有序的对象，序列里面的对象称为项。序列以‘[ ]’定义，每个项之间以逗号隔开。

一种创建序列的方法就是直接列出序列里面的项即可。每一个元素以逗号隔开，且放在[]之间。如下：

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

声明一个序列然后赋给weekDays变量。编译器知道我们是要创建字符串序列，因为每个项都是定义为字符串类型。 如果序列里面声明的是整数类型（var nums = [1,2,3];）编译器知道我们需要的是整数序列。

你也可以给序列一个特定的类型。

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

这么定义，告诉编译器weekDays接受一系列的String

序列里面还能定义序列

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

在这样的情况下，编译器会自动的平坦化序列，即转化为如下形式：

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

还有一种简短的方法来创建连续的数字。要创建1到100的序列，使用如下方法：

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

-使用布尔表达式创建序列

你能使用布尔表达式或者断言（predicate）来声明一个已存在的序列的子序列。例如，对于下面的序列：

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

接着，以第一个序列里面的项为基础，创建第二个序列，此序列只包含大于2的项。创建方法如下：

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

上面的表达式可以用语言描述为：“从num序列里选出所有大于2的项，然后将这些项赋给

numsGreaterThanTwo序列”。("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),"where"后面的"the value of an item is greater than 2"就是断言（predicate）

在这段代码里面：

1. 新创建的序列存放在numsGreaterThanTwo里面。

2. 代码： nums[n | n > 2];指定了源序列。在例子里面，nums就是已经存在的序列。

3. 遍历num里面的所有项，当表达式为true的时候，就返回这个项，由返回的所有项，创建一个新的序列。

4. 符号 "|"是用来分割变量n和后面的代码的。

5. 代码: nums[n | n > 2];定义了一个布尔表达式，它是是否要把原来序列里面的项拷贝到现在的序列里面去的一个衡量标准。

-访问序列元素

序列里的项可以通过下标来访问，从0开始。要访问一个元素，以序列的名字，后面跟" [元素的索引] "即可:

```
var 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]);
```

打印如下:

Mon
Tue
Wed
Thu
Fri
Sat
Sun

你还可以使用sizeof操作符后面跟上序列的名字来得到序列的长度：
sizeof days

下面的代码，打印7：

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

-向Sequence里插入项

insert关键字允许你向序列里的一个特定元素的前面或后面插入一个元素。

Note:事实上，序列是不变的。这意味着序列一旦被创建就不会改变。举个例子，当你插入或删除一项时，在此操作后，会创建一个新的序列并且这个序列会被赋给原来那个变量。

让我们来重新创建days序列,来证明一下：

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

这里，这个sequence只包含一个元素 "Mon".

我们可以使用insert和into关键字在序列的最后插入"Tue"。

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

类似的，我们添加 "Fri", "Sat"和"Sun"。

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

现在序列就包含了： "Mon", "Tue", "Fri", "Sat", and "Sun".

我们还可以使用insert和before关键字在给定的索引所指定的元素前面插入一项。记住，索引以0开始，所以"Fri"的索引是2.所以我们能够像下面这样在"Fri"前面插入"Thu".

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

现在序列包含了: "Mon", "Tue", "Thu", "Fri", "Sat", and "Sun".

在"Wed"后面插入"Tue",我们能使用insert和after关键字:

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

现在序列包含了一周的所有天： "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", and "Sun".

-从序列里删除项

delete和from关键字使得从序列里面删除项变得很简单。

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

现在序列包含："Mon", "Tue", "Wed", "Thu", "Fri", and "Sat".

你还可以删除一个特定索引的元素。下面的代码将从序列里面删除"Mon"(记住"Mon"是第一个元素，他的索引是0)

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

要删除序列里的所有项，使用delete跟上序列的名字即可：

```
delete days;
```

注意，delete只是把项从序列里面移走；它不会删除days这个变量。你还能够访问days变量，并向里面添加值。

-倒序序列里的项

你可以使用reverse操作符来倒序序列:

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

-比较序列

有时你可能会要比较序列是否相等。序列是按值来比较的：如果长度相同，里面的项也相同，那么他们就是相等的。

让我们来测试一下：

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

表达式seq1 == seq2将为true，因为序列拥有相同个数的项，且每个项的值都相同。所以，这段代码将打印true.

修改其中一个序列的项的个数，他们就不相同了：

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

这里输出"flase"，因为第二个序列比第一个序列长，所以两序列不相等。

我们还能通过修改项的值来使两个序列不相等，即使两个序列的长度仍然相等：

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

代码将会还是打印"false"，因为两个序列不等。

-使用序列片段

序列片段提供访问序列的一部分的功能。

```
seq[a..b]
```

这段语法将访问索引a和索引b之间的所有的元素。下面的脚本创建了一个只包含"Sat"和"Sun"的表示周末的序列。

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

```
seq[a..<b]
```

在索引a和b之间，且靠近b的地方使用"<"，将不包含索引b。我们可以用这个创建"Mon" 到"Fri"的表示工作日的序列。

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

```
seq[a..]
```

省略第二个索引，你能够访问从索引a到最后的所有项！举同样的例子，我们可以像下面这样创建表示周末的序

列：

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

```
seq[a..<]
```

最后，你能够使用"<"而不需要第二个索引，来访问序列a后面的，除了最后一项之外的所有项。

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

这里创建一个包含从 "Mon" 到 "Sat"的序列。

1.4 学习JavaFX脚本语言----6，7

发表时间: 2008-12-10

Lesson 6: Operators

目录：

- 连接操作符
- 运算操作符
- 一元操作符
- 关系操作符
- 条件操作符
- 类型比较操作符

-连接操作符

链接操作符"="是你最常用的操作符之一。使用"="，可以将其右边的值赋给左边的变量。

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

在前面的章节里面，你已经使用了很多次了。

-运算操作符

运算操作符提供加，减，乘，除的功能。"mod"操作符为求余！

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

下面提供一些例子：

```
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);
```

你还能同时使用运算操作符和连接运算符构成复合运算符。例如 result += 1; 和 result = result+1;两者的效果都是result加上了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);
```

只有"mod"操作符不能这么写。如果你想求一个数被2除的余数，你需要这么写：

```
result = result mod 2;
```

-一元操作符

大部分的操作符需要两个操作数；而一元操作符只需要一个操作数，对操作数进行加/减操作，取负数，如果操作数为boolean，则可以取逻辑反，等操作。

- 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

例子：

```
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
```

++/--操作符既可以在操作数的前面，也可以在操作数的后面。result++和++result运算的最后效果都是result加1。唯一的区别是result++是先取值后加1，++result是先加1再取值。如果你只是做简单的+/-，那么你用哪种方法都没有影响。但是，如果你在一个复杂的表达式里面使用，那么两种方法可能会不同。

下面的代码演示他们的区别：

```
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 prints 6!
println(result); // but the result is now 7
```

-关系操作符

关系操作符用来比较一个操作数是否大于，小于，等于或不等于另一个数。

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

下面的代码测试这些操作符：

```
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
```

-条件操作符

条件and和条件or是用在两个boolean表达式间的操作符。这两个操作符有"short-circuiting"的特性，就是第二个操作数只有在需要的时候才执行。举个例子,对于and操作符，如果第一个表达式的执行结果是false，那么第二个表达式就不会执行了，当第一个表达式为true的时候，第二个表达式才会执行。

and
or

下面是例子：

```
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：

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

-类型比较操作符

instanceof操作符判断一个对象是否是一个特定的类型。你可以使用此操作符来判断，一个对象是否是一个特定的类。

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

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

在学了以后的类和继承章节后，你会发现这个操作符很有用。

Lesson 7: 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

块表达式包含一系列的声明或表达式，由大括号包围，且以分号隔开。块表达式的值是块里面最后的表达式的值。如果块里面没有表达式，块表达式就是Void类型。注意，var和def是表达式。

下面的块表达式添加了一些成员并把结果存放在total变量里面：

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

输出：

Total is 24.

第一行 (var nums = [5, 7, 3, 9];) 声明了一个整型的序列。

接着声明了一个total变量来保存序列里面的值的总和。

块表达式，为大括号里面的所有代码：

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

在括号里面，第一行声明了一个变量sum，来保存序列里面的成员。第二行循环序列，取出序列里的成员与sum相加。最后一行，为返回块表达式的值。

-if表达式

if表达式表达式可以根据特定的条件执行特定的一段代码。

例如，下面的代码根据年龄设置票价。12-65岁票价10元。老人和小孩5元。5岁一下的免费。

```
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.");
```

年龄设为8时，输出如下：

Age: 8 Ticket Price: 5 dollars.

程序流程如下：

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

```
     ticketPrice = 10;
}
```

如果年龄小于5，票价为0.那么程序就会跳过其他的判断，直接打印出结果。
如果不小于5，那么程序跳转到第二个判断。

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

如果年龄在5-12岁，票价为5.
如果大于或等于12岁，就跳到了else语句里面。

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

这个块，只有在前面的条件都不符合的情况下才执行。设置12岁及以上的人的票价为12.

Note: 上面的代码可简写为：

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

你最好掌握，在后面的章节将会用到。

-范围表达式

在序列那一讲，你了解到了如果简单的定义一个有序的数字序列。

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

理论上，[0..5]就是个范围表达式。默认的，数字间递增1，但是你可以使用step关键字来修改递增的值。例如，定义一个1到10之间的所有奇数的序列。

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

输出

[1, 3, 5, 7, 9]

要创建一个递减的范围，请保证第二个数要小于第一个数，并且要将step设为一个负数。

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

输出：

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

如果你创建了一个递减的序列，但是没有设定step的值，你将会得到一个空的序列。

下面的代码：

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

将会在编译的时候出现警告：

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

如果你忽略了警告，那你得到的是个空的序列。

-for表达式

另一个和序列有关的表达式是for表达式。for表达式提供一个简便的循环结构来遍历序列里的所有项。

下面是个例子:

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

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

输出:

Mon
Tue

Wed
Thu
Fri
Sat
Sun

让我们来分析这段代码。for表达式以for开头

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

days变量是序列的名称，for表达式将遍历此序列。

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

day变量依次持有序列里的项

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

注意，day变量并不需要声明就可以在for表达式里面使用了。并且，for表达式结束后，day无法被访问。程序员经常给临时变量一个简短的名字以示区别。

在前面的例子里面，for没有显示的返回值。但是for返回一个序列。下面的两个例子来使用for来从一个序列创建另一个序列。

```
// 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();
```

注意了，toUpperCase方法由String提供。你能在API里找到完整定义。

-while表达式

另一个循环表达式是while表达式。和for不同，不是遍历序列的，而是根据条件来循环的。while是一个语义(syntactically)上的表达式，为Void类型，没有返回值。

给个例子：

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

输出：

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

count == 9

第一行声明了一个变量count并初始化为 0

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

接着开始while表达式。表达式循环，知道count为10为止。

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

循环体中，打印count值，并加 1 。

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

当count为10时，循环结束。如果要创建一个无限循环，条件里改为true关键字，就像这样： while(true){}

-break和continue表达式

break和continue表达式和循环表达式有关。这两个表达式是作用于循环上的：break终止循环，continue跳过当前的循环。

brean和continue是语义(syntactically)表达式。他们为Void类型，没有返回值。

例子：

```
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，程序将会简单的输出：0到10。

如果只有第一个if表达式，程序将在i比5大的时候跳出循环。

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

程序将只打印1到5。

加上第二个if后，程序会间隔的执行循环。

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

这里，表达式只有在i为偶数的时候才执行continue。当continue执行的时候，println()将不会被执行，也就不会打印了。

-throw, try, catch 和 finally表达式

在现实世界的应用里面，有时可能会出现不正常的程序流程。例如，一段脚本读取文件，但是文件找不到，那么这段脚本就无法执行了。我们将这种情况称为"异常"。

Note:异常是对象。一般可以从字面意思上知道他们表示的是什么异常。（例如，FileNotFoundException表示了无法找到文件。）但是，下面的例子并不是来展现各种各样的异常的。所以，以普通的异常来说明throw, try, catch, 和 finally

下面的脚本定义一个方法并抛出异常：

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

运行，输出：

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

但是，将变量改为true，异常将被抛出。在运行时，脚本将会中断，并打印如下信息：

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

要阻止程序被异常终端，我们需要用try/catch来捕获foo()抛出的异常。由名字来看，这个表达式的作用是，try执行一些代码，但是发生了问题就catch到了一个异常。

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

现在，程序打印：

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

还有个finally块（这个不能算得上是个表达式），不论是否有异常，finally块里的代码都会被执行。finally块一般是用来做清理工作的。

```
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：

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

1.5 学习JavaFX脚本语言----8，9

发表时间: 2008-12-11

Lesson 8: Data Binding and Triggers

目录

- 绑定概述
- 绑定和对象
- 绑定和方法
- 绑定序列
- 替换触发器

-绑定的概念

bind关键字将目标变量的值和一个范围表达式联系(bound expression)起来。范围表达式可以是基本类型，一个对象，方法的返回值或者一个表达式的返回值。
下面的章节将一个个的举例。

-绑定和对象

现实中，大部分情况下，你要使用数据绑定，来同步GUI和它的数据
(GUI是 《Building GUI Applications with JavaFX》的主题; 下面我们演示的是简单的非GUI例子)

我们从简单的开始：下面的脚本中，将变量x绑定到了变量y上，改变x的值，然后打印出y的值。由于变量被绑定了，y的值会自动的更新为新值。

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

注意，我们是用def声明的变量y。这样的话，能够阻止直接修改y的值！（y的值允许以绑定的方式修改）在下

面的绑定对象的例子中，你应该以相同的方式来绑定。

```
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}");
```

当你修改myStreet的值的时候，address对象里的street变量的值也会随之改变。

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

注意，myStreet值的改变将导致一个新的Address对象被创建，并且这个对象被赋予了address变量。如果想改变值，但是不要新创建一个新的Address对象，直接在实例变量上进行绑定。

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

```
     zip: bind myZip;
};
```

如果你已明确的对实例变量进行了绑定，那么你可以省略第一个bind（Address前面的那个bind）

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

-绑定和方法

前面已经讨论过方法了，但是你还要学习一下bound functions 和non-bound functions之间的差别。

下面的方法，它创建并返回一个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bound function，因为它以bound关键字开头。

Note: bound关键字不是替代bind关键字的。在下面的例子里面，将会同时使用bound和bind.

接着，我们添加一些代码来调用这个方法并测试绑定：

```
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);
```

输出:

3.0
10.0
20.0

我们来分析这段代码：

代码:

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

初始化变量myX和myY为3.0。这些值将已参数的方式传给makePoint方法。makePoint方法前面的bind关键字，将新创建的Point对象pt绑定到了makePoint方法的返回值上。(
The bind keyword, placed just before the invocation of makePoint, binds the newly created Point object (pt) to the outcome of the makePoint function.)[这里字面意思是pt绑定到了makePoint的返回值上去了，我理解的是返回值绑定到pt上去。]

下一段代码:

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

修改myX的值为10.0然后打印pt.x。pt.x也将输出10.0

最后一段代码:

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

修改scale的值并打印出pt.x。现在pt.x的值是20.0，如果我们将方法前面的bound关键字去掉（就变成了non-bound function），输出将会是：

```
3.0
10.0
10.0
```

这是因为non-bound functions只有在它的参数发生改变的时候才会重新执行。而scale并不是方法的参数，所以改变scale的值不会对方法产生影响。

-绑定序列

你还可以在表达式上使用bind。要解释这一点，让我们先来定义两个序列并打印他们的值。

```
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有10个项（1到10）.seq2也有10个项，并且和seq1的值相等,但是我们给每项都乘以了2，所以值将是seq1的两倍。

输出：

第一个序列:
1
2
3
4
5
6
7
8
9
10
第二个序列:
2
4
6
8
10
12
14
16
18
20

只要在for关键字前面加上bind关键字，我们就能将两个序列绑定了。

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

现在的问题就是“如果seq1里面发生改变了，seq2里面的所有项或部分项会有影响吗？”我们可以在seq1的末尾插入一项（值为11）来测试一下。

```
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:

第一列:
1
2
3
4
5
6
7
8
9
10
11
第二列:
2
4
6
8
10
12
14
16
18
20

22

输出显示了在seq1中插入11后，对seq2里的前面10项并没有产生影响，而是自动的加到了seq2的后面，并且值是22.

-替换触发器（replace trigger）

替换触发器是一段特殊的代码，它和变量关联，并且只要变量的值发生改变就会执行。下面的例子展示了基本的语法：定义了一个password变量并和一个替换触发器关联，当password改变的时候，触发器打印出password的新值。

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

password = "bar";
```

输出:

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

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

这里触发器被触发了两次：第一次，当password被初始化为"foo"的时候，以及第二次当值变成"bar"时。
注意，oldValue变量持有触发器执行前变量的值。你能够任意命名此变量，我们使用oldValue只是因为它比较直观。

Lesson 9: Writing Your Own Classes

目录

- 例子：Customer
- 从其他类继承

-例子：Customer

在Writing Scripts章节，你学会了如何使用对象。但是，当时我们是让你去下载了.class文件，以使编译器知道怎么去创建Address和Customer对象。在下面的例子里面，我们重新来看代码，新增缺少的类定义，以保证所有的代码都能通过编译。

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

如果你已经掌握了变量和方法，那么这段代码你应该很容易理解。Address类声明了street, city, state, and zip实例变量，且都是String类型。Customer也声明了一些实例变量还有一些方法来打印他们的值。因为这些变量和方法都是在类里面声明的，所以在你创建的任意的Address和Customer类的实例都能够访问到他们。

-从其他类继承

你还能创建一个类，并从其他类里面继承变量和方法。例如，假设在银行里面保存，验证一个帐户。每个帐户都有帐户号码和金额。你能查询金额，存款或取款。我们能对此建模，抽象出一个基本的帐户类并且给与通用的变量和方法：

```
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;
    }
}
```

我们将此类标记为抽象的，这样的话，Account对象是不能够直接被创建出来的（继承的类只需要savings accounts or checking accounts）

accountNum和balance 变量持有帐户号码和当前的帐户金额。 而余下的那些方法则提供了基本的取钱，存钱和查询金额的功能。 T

我们能定义一个SavingsAccount，并使用extends关键字来继承得到这些变量和方法。

```
class SavingsAccount extends Account {

    var minBalance = 100.00;
    var penalty = 5.00;

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

```
    }
}
```

因为SavingsAccount是Account的子类，所以它将自动的包含Account里面所有的实例变量和方法。这可以使我们专注于SavingsAccount所特有的属性和方法（比如，如果帐户金额小于100，则无法取款）

类似的，我们再定义一个CheckingAccount类继承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!
        }
    }
}
```

这里定义了一个变量，来判断帐户持有者能否透支。(如果取款的时，取款金额超过了帐户现有金额，那么透支功能将起作用，判断此用户是否能透支)注意，在这里我们修改了继承的withdraw方法，这就是方法的覆盖，所以方法前面需要加上override关键字

1.6 学习JavaFX脚本语言----10，11（完）

发表时间: 2008-12-12

Lesson 10: Packages

目录

- Step 1: 选择一个包名
- Step 2: 创建目录
- Step 3: 添加包声明
- Step 4: 添加访问权限
- Step 5: 编译源码
- Step 6: 使用类

到这里，你对javaFX的基础应该比较熟悉了。但是对于源文件的存放，你可能还不是很清楚（你现在可能是用的一个文件夹来存放所有的例子代码）我们将把代码放到包中，来改变你对存放代码的认识。（We can improve our overall organization by placing our code into packages.）

包能够让你按功能来将代码分类保存。它还给你的类一个唯一的命名空间。我们将在下面一步步的来将Address类存放到一个特殊的包内。

-Step 1: 选择一个包名

在我们修改代码前，我们需要给包起个名字。由于我们的Address类是假设用在addressbook应用上的，所以我们使用"addressbook"作为包名。

-Step 2: 创建目录

接着，我们必须创建一个addressbook目录。这个目录里面将包含所有我们设计的属于addressbook这个包的.fx文件。你可以在任意的地方创建目录。我们在例子里面使用 /home/demo/addressbook，但是脚本必须在一个和包名相同的目录里面，这里就是 addressbook

-Step 3: 添加包声明

现在，到addressbook目录里面创建Address.fx源代码文件。粘贴下面的代码到源代码文件里面去。第一行提

供了一个包声明，这将表示这个类属于addressbook这个包。

```
package addressbook;

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

注意了，如果源代码里面有包声明，它必须在其它代码的前面，即源码文件的第一行。没个源文件只能有一个包声明。

-Step 4: 添加访问权限

接下来，我们需要在Address的类和变量上加上pulic关键字。

```
package addressbook;

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

这个关键字是5个访问限制修饰符里面的一个。我们会在下一节介绍访问限制修饰符。现在你只要知道，public关键字使得这段代码能够被其他的类和脚本访问。

-Step 5:编译源码

依然在addressbook目录里面，像平时一样使用javafxc Address.fx命令编译即可。编译完成后，这个文件夹里面将包含编译得到的.class文件。

-Step 6: 使用类

现在我们能够测试修改后的Address类了。但是我们必须先返回到父目录 /home/demo。这里我们创建一个简单的脚本packagetest.fx类测试如何使用addressbook包。

我们有两种方法来访问这个类：

```
// Approach #1

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

Approach #1使用完整的类名创建了一个对象（addressbook.Address）。对比另一种方法，这种方法比较的笨拙，但是你还是需要知道有这种写法。

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

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

Approach #2使用import关键字，import关键字允许你以简短的名字在脚本里面使用类。当程序比较大时，推荐使用这种方法，因为它是self-documenting。

Lesson 11: Access Modifiers

目录

- 默认访问权限
- package访问权限修饰符
- protected访问权限修饰符
- public访问权限修饰符
- public-read访问权限修饰符
- public-init访问权限修饰符

-默认访问权限

当你不提供任何访问权限控制符的时候，就是默认的访问权限，也就是"script-only".这也是我们在教程里面最常使用的权限。

例子:

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

这一级别的访问权限使得变量只能在脚本内被initialized, overridden, read, assigned, 或 bound.其他文件无法访问。

-package访问权限修饰符

为了让变量，方法或类能被包里的其他代码访问到，使用package访问权限修饰符

```
package var x;
```

不要把这和前一节包的声明玳瑁搞混淆。

例子:

```
// 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();
```

你能使用下面的命令编译和运行这个例子：

javafxc tutorial/one.fx tutorial/two.fx
javafx tutorial/two

输出：

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

-protected访问权限修饰符

protected访问权限修饰符使得变量或方法不仅能时包里的其他代码访问到，还能使不在同一个包内的子类访问到。

例子:

```
// 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();
```

编译运行：

javafxc tutorial/one.fx two.fx
javafx two

输出：

Class two says Hello!

Note:这个访问权限修饰符不能用在类上，这就是为什么我们在类one前面写的是public的原因。

-public访问权限修饰符

一个pulibc的类，变量，方法具有最大可见度，即，它可以被任意的类或脚本访问，不管是不是在一个包内。

例子:

```
// 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();
```

编译运行：

javafxc 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访问权限修饰符

public-read访问权限修饰符修饰的变量是公共的但是对外是只读的，只能被当前的脚本修改。如果想扩大它的可被修改的权限范围，在前面加上package或protected（ackage public-read 或者 protected public-read）这样的话，可以使得package或protected范围的代码能修改此变量。

例子:

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

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

编译运行：

javafxc tutorial/one.fx tutorial/two.fx
javafx tutorial/two

输出是"1"，这证明了x能够被tutorial/one.fx外的其他代码读取到。

现在我们来试着修改它的值:

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

结果是编译期错误：

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

```
  ^
1 error
```

为了让此代码能运行，我们要扩大x的写的访问权限：

```
// 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);
```

现在将打印"2"。

-public-init访问权限修饰符

public-init访问权限修饰符修饰的变量能被任何包里的对象初始化。初始化后的写操作权限限制，却是和public-read类似的访问控制。（默认是脚本级别的写操作，前面加上package或 protected就扩大了访问权限了）

例子:

```
// 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：

javafxc tutorial/one.fx two.fx
javafx two

打印出"Initialized this variable from a different package!", 这证明了其他包内的对象能够初始化message变量。但是，接下来的写操作权限却是script-only，我们不能修改它的值。

```
// 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);
```

编译出错：

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

这证明了这个很特别的行为：对象能够被任意对象初始化，但是初始化以后却是受不同的访问级别控制的。

ps:呼！花了一周时间，终于翻译完了！还好语法比较简单，翻译难度不大！如果有时间和精力，还会翻译《Building GUI Applications With JavaFX》。