精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

级别： 初级

Andrew Glover (aglover@stelligent.com), 总裁, Stelligent Incorporated

2008 年 4 月 21 日

本教程适合于不熟悉 Groovy，但想快速轻松地了解其基础知识的 Java™ 开发人员。了解 Groovy 对 Java 语法的简化变形，学习 Groovy 的核心功能，例如本地集合、内置正则表达式和闭包。编写第一个 Groovy 类，然后学习如何使用 JUnit 轻松地进行测试。借助功能完善的 Groovy 开发环境和使用技能，您将轻松完成本教程的学习。最重要的是，您将学会如何在日常 Java 应用程序开发中联合使用 Groovy 和 Java 代码。

开始之前

了解本教程的主要内容，以及如何从中获得最大收获。

关于本教程

如果现在有人要开始完全重写 Java，那么 Groovy 就像是 Java 2.0。Groovy 并没有取代 Java，而是作为 Java 的补充，它提供了更简单、更灵活的语法，可以在运行时动态地进行类型检查。您可以使用 Groovy 随意编写 Java 应用程序，连接 Java 模块，甚至扩展现有的 Java 应用程序 — 甚至可以用 Groovy 对 Java 代码进行单元测试。Groovy 的美妙之处还在于，它能够比编写纯粹的 Java 代码更快地完成所有工作 — 有时候会快许多。

在本教程中，您将了解到 Groovy 是一门动态语言，它能够像 Java 语言本身一样很好地应用于 Java 平台。

学习目标

本教程将逐步向您介绍 Groovy 的基本概念。您将学习 Groovy 集合、Groovy 类，当然还有 Groovy 的语法。完成本教程之后，您将了解将 Java 和 Groovy 结合使用的好处，从此您将能够在日常的 Java 开发中使用 Groovy。

前提条件

为了从本教程得到最大收获，您应该熟悉 Java 语法和在 Java 平台上进行面向对象开发的基本概念。

系统需求

要尝试本教程的代码，需要安装以下环境之一：

- Sun 的 JDK 1.5.0_09（或更新版本）或
- IBM developer kit for Java technology 1.5.0 SR3

另外，本章教程假设您正在使用 Eclipse IDE。不需要安装 Groovy，因为本教程会介绍如何安装 Groovy Eclipse 插件。

本教程推荐系统的配置如下：

- 支持 Sun JDK 1.5.0_09（或更高版本）或 IBM JDK 1.5.0 SR3 的系统，拥有至少 500 MB 主内存
- 至少 20 MB 可用硬盘空间，用来安装本教程涉及的软件组件和示例

本教程的操作说明和示例均基于 Microsoft Windows 操作系统。本教程涉及的所有工具在 Linux 和 Unix 系统上也能工作。

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

关于 **Groovy**

这一节将学习 Groovy 的基础知识：它是什么，它与 Java 语言和 JVM 的关系，以及编写 Groovy 代码的一些要点。

什么是 Groovy?

Groovy 是 JVM 的一个替代语言 — *替代* 是指可以用 Groovy 在 Java 平台上进行 Java 编程，使用方式基本与使用 Java 代码的方式相同。在编写新应用程序时，Groovy 代码能够与 Java 代码很好地结合，也能用于扩展现有代码。目前的 Groovy 版本是 1.5.4，在 Java 1.4 和 Java 5 平台上都能使用，也能在 Java 6 上使用。

Groovy 的一个好处是，它的语法与 Java 语言的语法很相似。虽然 Groovy 的语法源于 Smalltalk 和 Ruby 这类语言的理念，但是可以将它想像成 Java 语言的一种更加简单、表达能力更强的变体。（在这点上，Ruby 与 Groovy 不同，因为它的语法与 Java 语法差异很大。）

许多 Java 开发人员非常喜欢 Groovy 代码和 Java 代码的相似性。从学习的角度看，如果知道如何编写 Java 代码，那就已经了解 Groovy 了。Groovy 和 Java 语言的主要区别是：完成同样的任务所需的 Groovy 代码比 Java 代码更少。（有时候会少很多！）

Groovy 快捷方式

开始使用 Groovy 时，您会发现它使日常的编程活动变得快了许多。完成本教程之后，您会了解更多的 Groovy 语法快捷方式。不过现在只需知道以下这些要点：

- Groovy 的松散的 Java 语法允许省略分号和修改符。
- 除非另行指定，Groovy 的所有内容都为 `public`。
- Groovy 允许定义简单脚本，同时无需定义正规的 `class` 对象。
- Groovy 在普通的常用 Java 对象上增加了一些独特的方法和快捷方式，使得它们更容易使用。
- Groovy 语法还允许省略变量类型。

Groovy 的新增特性

虽然 Groovy 允许省略 Java 语法中的一些元素，但也增加了一些新特性，例如本地集合、内置的正则表达式和闭包。在标准的 Java 代码中，如果想要创建一个项列表，首先要导入 `java.util.ArrayList`，然后程序化地初始化 `ArrayList` 实例，*然后* 再向实例中添加项。在 Groovy 中，列表和映射都内置在语法中 — 无需导入任何内容。正则表达式也不需要额外的导入或对象；它们可以通过特殊的 Groovy 语法来创建。

关于闭包

对于任何 Java 开发人员来说，闭包都是一个令人兴奋的新技巧。这些神奇的构造将会包含在未来的 Java 发行版（很可能是 Java 7）中，成为正式的 Java 语法，但现在已经可以在 Groovy 中使用了。可以将*闭包* 想像为一个代码块，可以现在定义，以后再执行。可以使用这些强大的构造做许多漂亮的事，不过最著名的是简化迭代。使用 Groovy 之后，就有可能再也不需要编写 `Iterator` 实例了。

动态的 Groovy

从技术上讲，Groovy 可能是您最近听说过的类型最松散的动态语言之一。从这个角度讲，Groovy 与 Java 语言的区别很大，Java 语言是一种固定类型语言。在 Groovy 中，类型是可选的，所以您不必输入 `String myStr = "Hello";` 来声明 `String` 变量。

除此之外，Groovy 代码还能在运行时轻松地改变自己。这实际上意味着，能够在运行时轻松地为对象指定新方法和属性。这一编程领域称为*元编程*，Groovy 能够很好地支持这种编程方式。在学习本教程的过程中，您将了解到关于 Groovy 的动态性质的更多内容。现在惟一要补充的是，您会惊讶地发现，在 Groovy 会使操作 XML 或普通的 `java.io.File` 实例变得非常轻松。

一体两面

用 Groovy 编写的任何内容都可以编译成标准的 Java 类文件并在 Java 代码中重用。类似地，用标准 Java 代码编写的内容也可以在 Groovy 中重用。所以，可以轻易地使用 Groovy 为 Java 代码编写单元测试。而且，如果用 Groovy 编写一个方便的小工具，那么也可以在 Java 程序中使用这个小工具。

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

Groovy 初探

学习新语言并不是件小事，即使是 Groovy 也不例外。这一节将介绍学习 Groovy 的更多动力。另外还将第一次看到一些 Groovy 代码，并了解 Groovy 与 Java 编程的比较。

为什么要学习 Groovy?

即使 Groovy 与 Java 语言有许多相似之处，它仍然是另一个语言。您可能想知道为什么应该花时间学习它。简单的回答就是：Groovy 是一种更有*生产力*的语言。它具有松散的语法和一些特殊功能，能够加快编码速度。

只用一个示例即可说明问题：一旦发现使用 Groovy 在集合中导航的容易程度，您就再也不会用 Java 处理集合导航了。能够用 Groovy 快速编写代码，这还意味着能够更快地收到反馈，更不用说完成任务列表中的工作带来的满足感了。在较高层面上，如果能更快地将代码交付给利益相关者，那么就能在更短的时间内交给他们更多发行版。实际上，Groovy 比 Java 更有助于敏捷开发。

入门非常容易

如果仍然觉得采用新语言很困难，那么可以看看将 Groovy 集成到开发环境有多么容易。您无需安装新的运行时工具或专门的 IDE。实际上，只需将 Groovy 的一个 jar 文件放在类路径中即可。

而且，Groovy 是一种开源语言，由热心的 Java 开发人员社区管理。因为 Groovy 获得 Apache Software License, Version 2.0，所以可以自由地使用它开发自由软件和私有软件。

Groovy 和 Java 语言对比

买车的时候，如果不试驾一下，是不会买的。所以，在要求您安装 Groovy 之前，我会演示一些代码。首先，回顾一下用 Java 如何创建、编译和运行标准的 Hello World 示例；然后再看看如何使用 Groovy 代码执行同一过程。比较这两个示例，很容易就能看到这两种语言之间的差异。

用 Java 编写的 Hello World

用 Java 编写的典型的 Hello World 示例如下所示：

```
public class HelloWorld {
  public static void main(String[] args) {
        System.out.println("Hello World!");
  }
}
```

编译和运行 Java 示例

在这个简单的 `HelloWorld` 类中，我省略了包，而且向控制台输出的时候没有使用任何多余的编码约定。下一步是用 `javac` 编译这个类，如下所示：

```
c:>javac HelloWorld.java
```

最后，运行经过编译的类：

```
c:>java HelloWorld
```

迄今为止还不错 — 很久以前就会编这么基础的代码了，所以这里只是回顾一下。下面，请看用 Groovy 编码的相同过程。

用 Groovy 编写的 Hello World

就像前面提到过的，Groovy 支持松散的 Java 语法 — 例如，不需要为打印 “Hello World!” 这样的简单操作定义类。

而且，Groovy 使日常的编码活动变得更容易，例如，Groovy 允许输入 `println`，而无需输入 `System.out.println`。当您输入 `println` 时，Groovy 会非常聪明地知道您指的是 `System.out`。

所以，用 Groovy 编写 Hello World 程序就如下面这样简单：

```
println "Hello World!"
```

请注意，在这段代码周围没有类结构，而且也没有方法结构！我还使用 `println` 代替了 `System.out.println`。

运行 Groovy 示例

假设我将代码保存在文件 MyFirstExample.groovy 内，只要输入以下代码就能运行这个示例：

```
c:>groovy MyFirstExample.groovy
```

在控制台上输出 “Hello World!” 所需的工作就这么多。

快捷方式应用

您可能注意到了，我不必编译 `.groovy` 文件。这是因为 Groovy 属于*脚本语言*。脚本语言的一个特点就是能够在运行时进行解释。（在 Java 中，要从源代码编译生成字节码，然后才能进行解释。区别在于脚本语言能够*直接*解释源代码。）

Groovy 允许完全省略编译步骤，不过仍然*可以*进行编译。如果想要编译代码，可以使用 Groovy 编译器 `groovyc`。用 `groovyc` 编译 Groovy 代码会产生标准的 Java 字节码，然后可以通过 `java` 命令运行生成的字节码。这是 Groovy 的一项经常被忽略的关键特性：用 Groovy 编写的所有代码都能够通过标准 Java 运行时编译和运行。

至于运行代码，如果我希望更加简洁，我甚至还能输入

```
c:>groovy -e "println 'Hello World!'"
```

这会生成相同的结果，而且甚至无需定义任何文件！

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

Groovy 入门

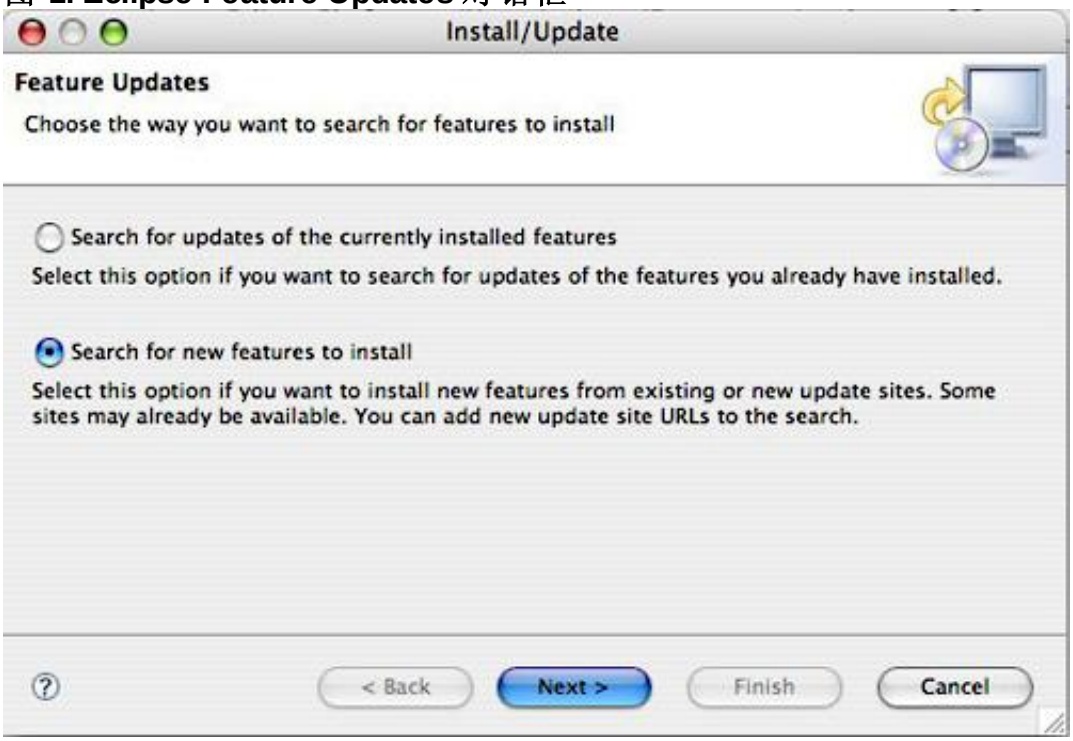
在这一节中，将真正开始进行 Groovy 编程。首先，学习如何轻松地安装 Groovy（通过 Eclipse Groovy 插件），然后从一些有助于了解 Groovy 的简单示例开始。

轻松安装 Groovy

为了迅速开始使用 Groovy，需要做的全部工作就是安装 Eclipse 的 Groovy 插件。打开 Ecliplse，在 **Help** 菜单中选择 **Software Updates** > **Find and Install...**。

图 1 显示了执行以上步骤之后出现的对话框：

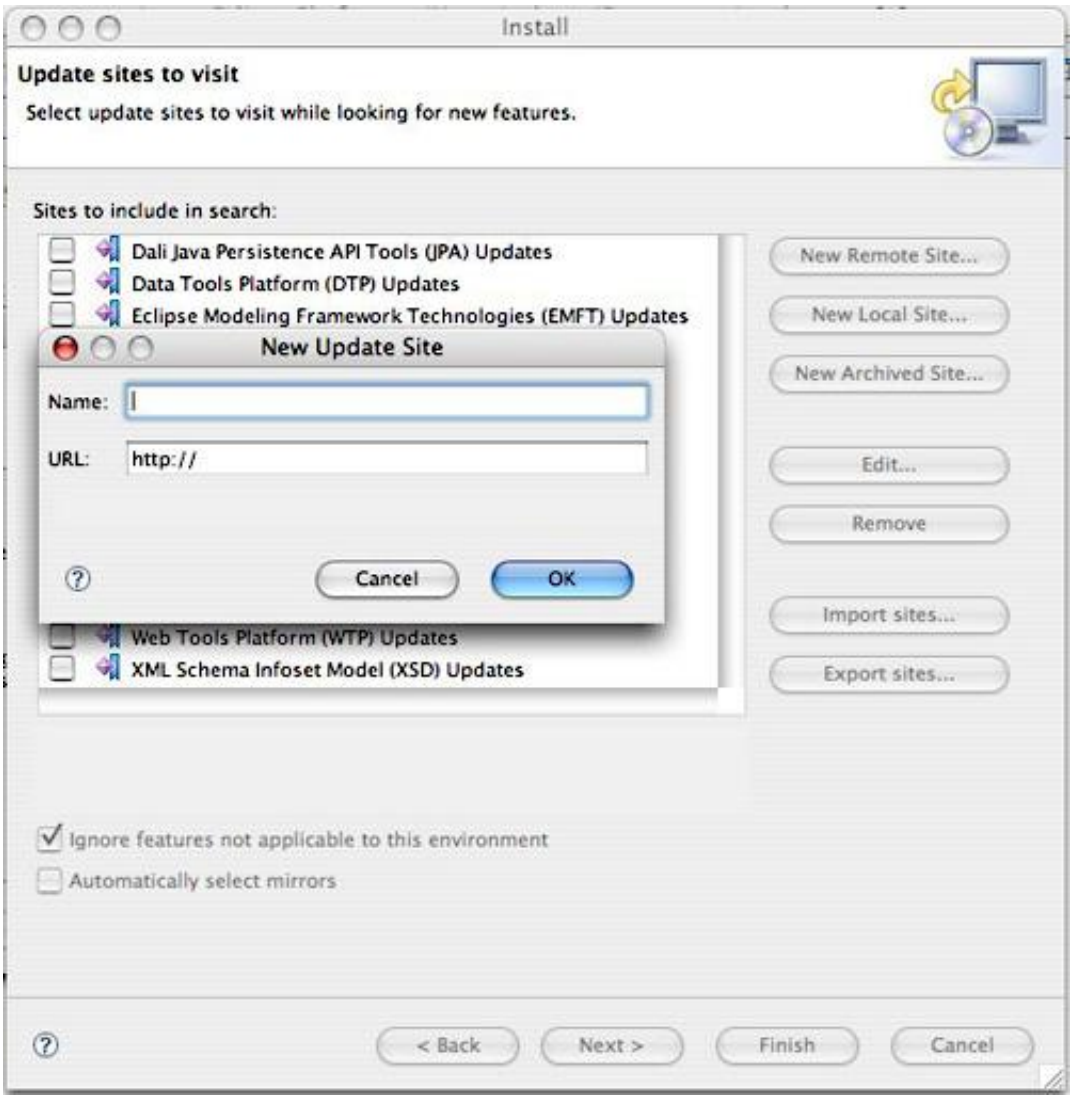
图 1. Eclipse Feature Updates 对话框



在选项中导航

接下来，出现一个对话框，里面包含两个选项。请选择 **Search for new features to install** 单选按钮。单击 **Next** 按钮，然后选择 **New Remote Site...**。出现一个新的对话框，里面包含两个需要填写的字段：新位置的名称和该位置的 URL，如图 2 所示：

图 2. 确保为新的远程站点提供了正确的 URL



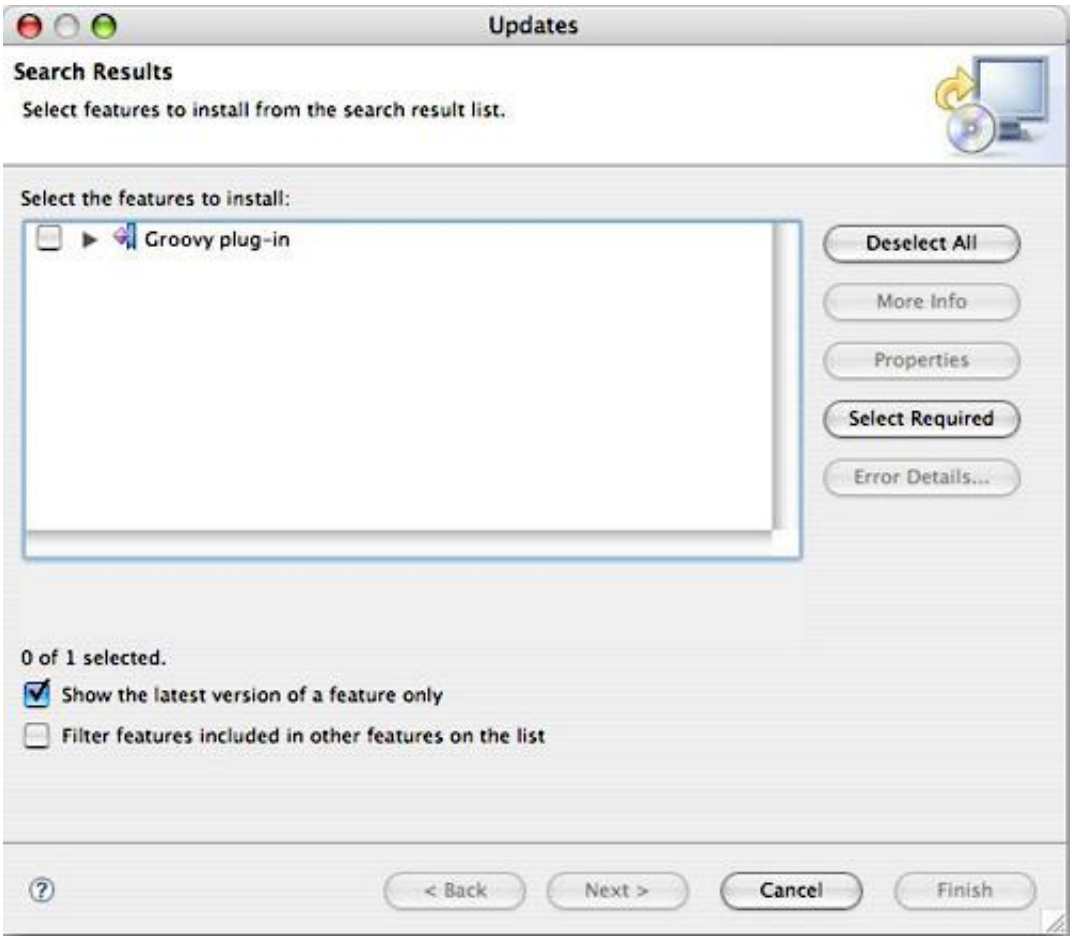
输入 “Groovy plugin” 作为名称，输入 “http://dist.codehaus.org/groovy/distributions/update/” 作为位置，单击 **OK** 按钮，在随后出现的 **Sites to include in search** 框中确保选中了名为 “Groovy plugin” 的项目 — 现在的列表应该如图 3 所示。

图 3.Eclipse 中的远程网站清单

完成安装

单击 **Finish** 按钮之后，应该会出现 Search Results 对话框。请再次确定选中了 “Groovy plugin” 框并单击 **Next** 按钮，这一步骤如图 4 所示：

图 4. 选择 Groovy 插件

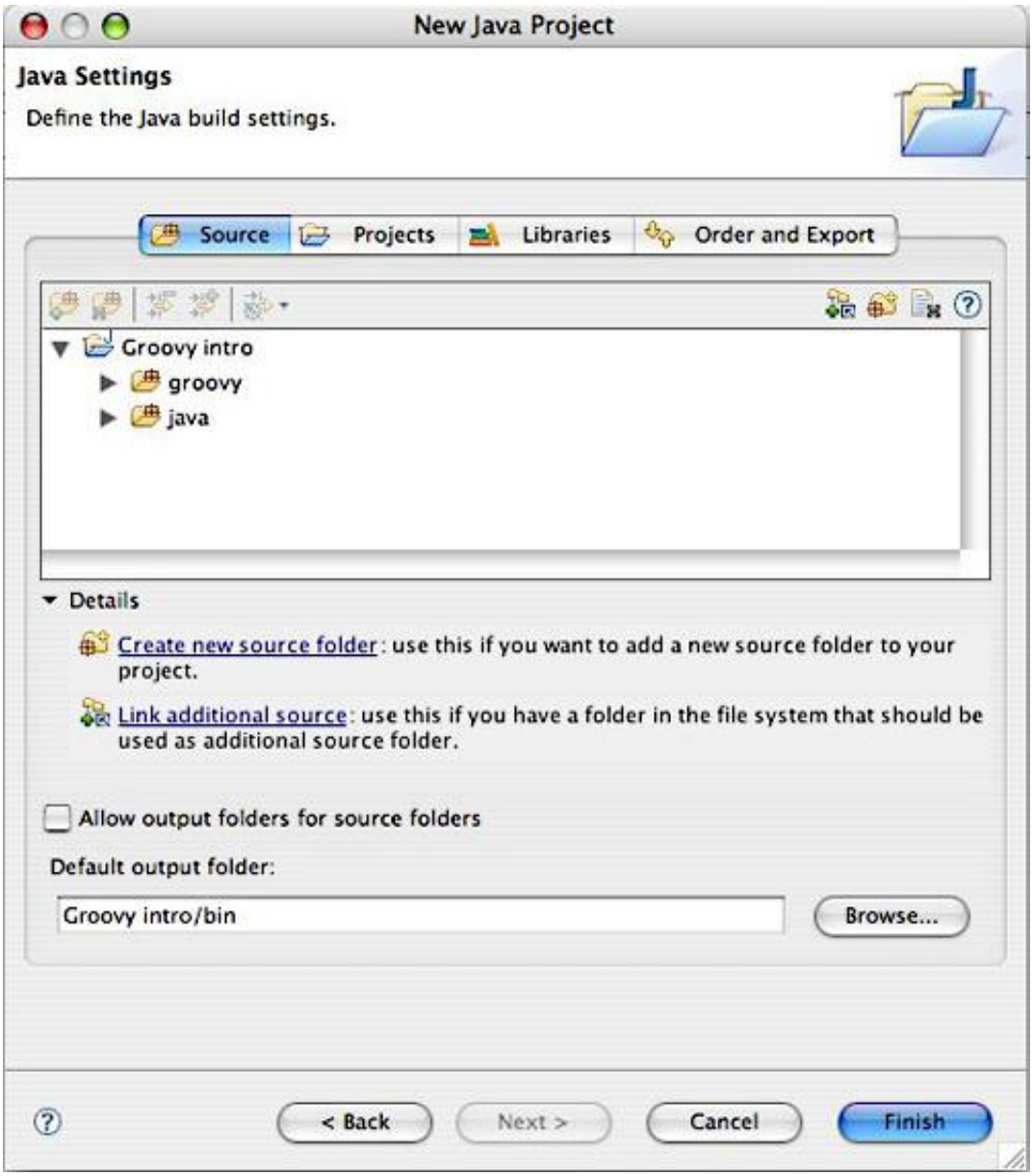


经过一系列确认之后，将会下载插件，然后可能需要重新启动 Eclipse。

创建 Groovy 项目

Eclipse 重启之后，就能够创建第一个 Groovy 项目了。请确保创建两个源文件夹 — 一个称为 “groovy”，另一个称为 “java”。编写的 Groovy 代码放在 groovy 文件夹，Java 代码放在 java 文件夹。我发现将二者分开将会很有用，如图 5 所示：

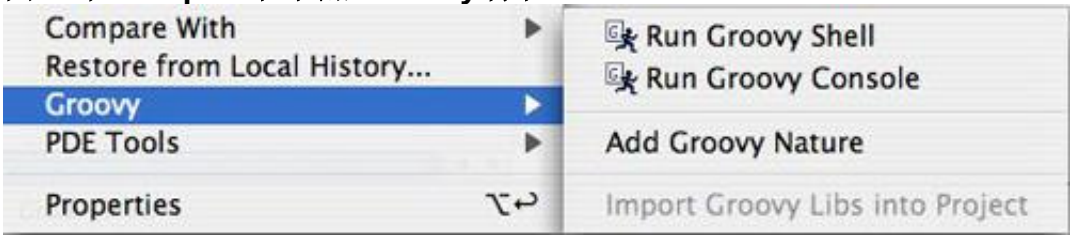
图 5. 两个源文件夹 — Java 和 Groovy



将 Groovy 导入项目

项目创建之后，右键单击项目的图标，应该会看到一个 **Groovy** 选项，如图 6 所示。请选择该选项，然后选择 **Add Groovy Nature** 选项。这样做可以将必要的 Groovy 库、编译器和运行程序导入到项目中。

图 6. 在 Eclipse 中添加 Groovy 特性

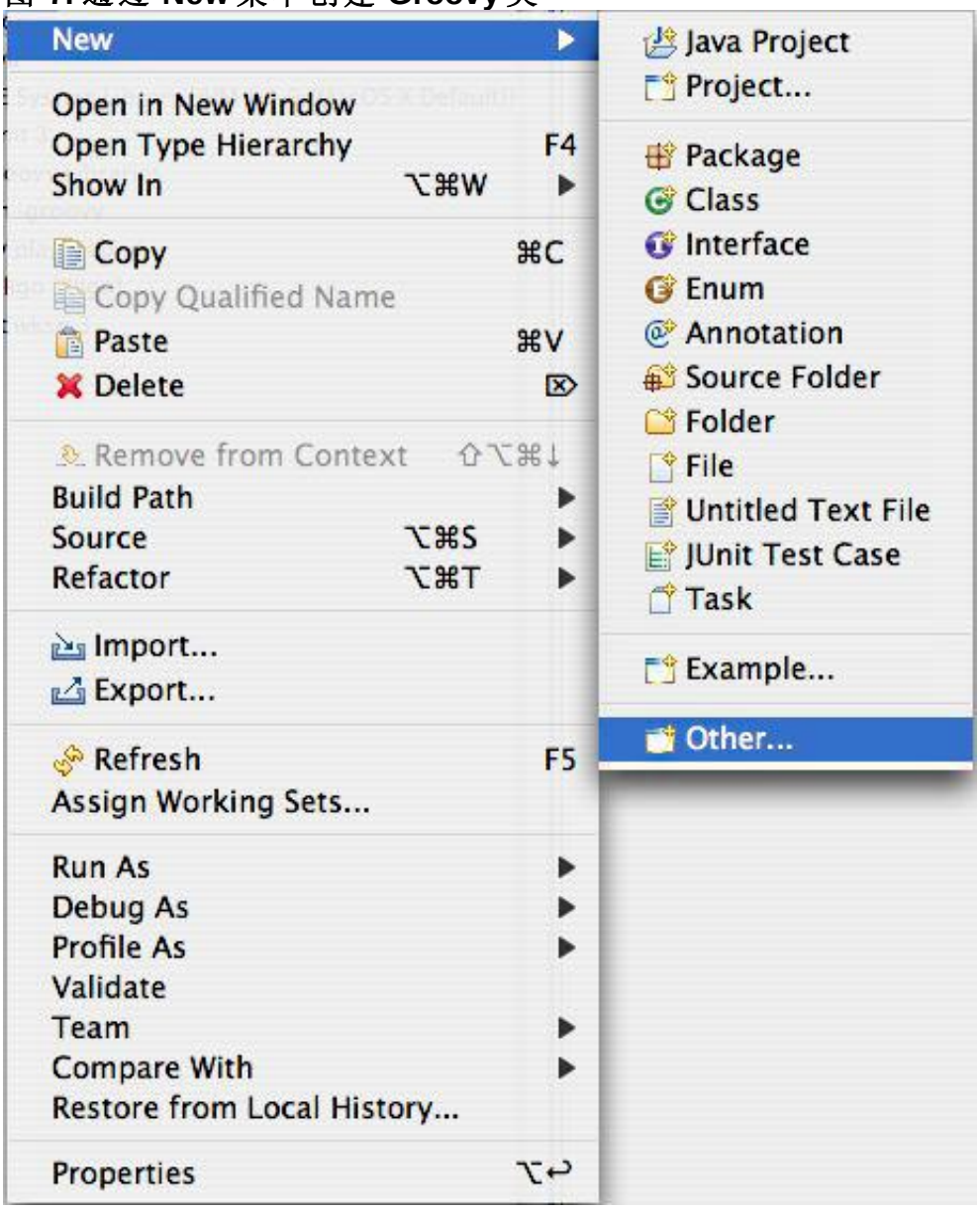


创建 Groovy 类

创建 Groovy 类很简单。选择 `groovy` 文件夹并右键单击它。选择 **New**，然后选择 **Other**，如图

7 所示：

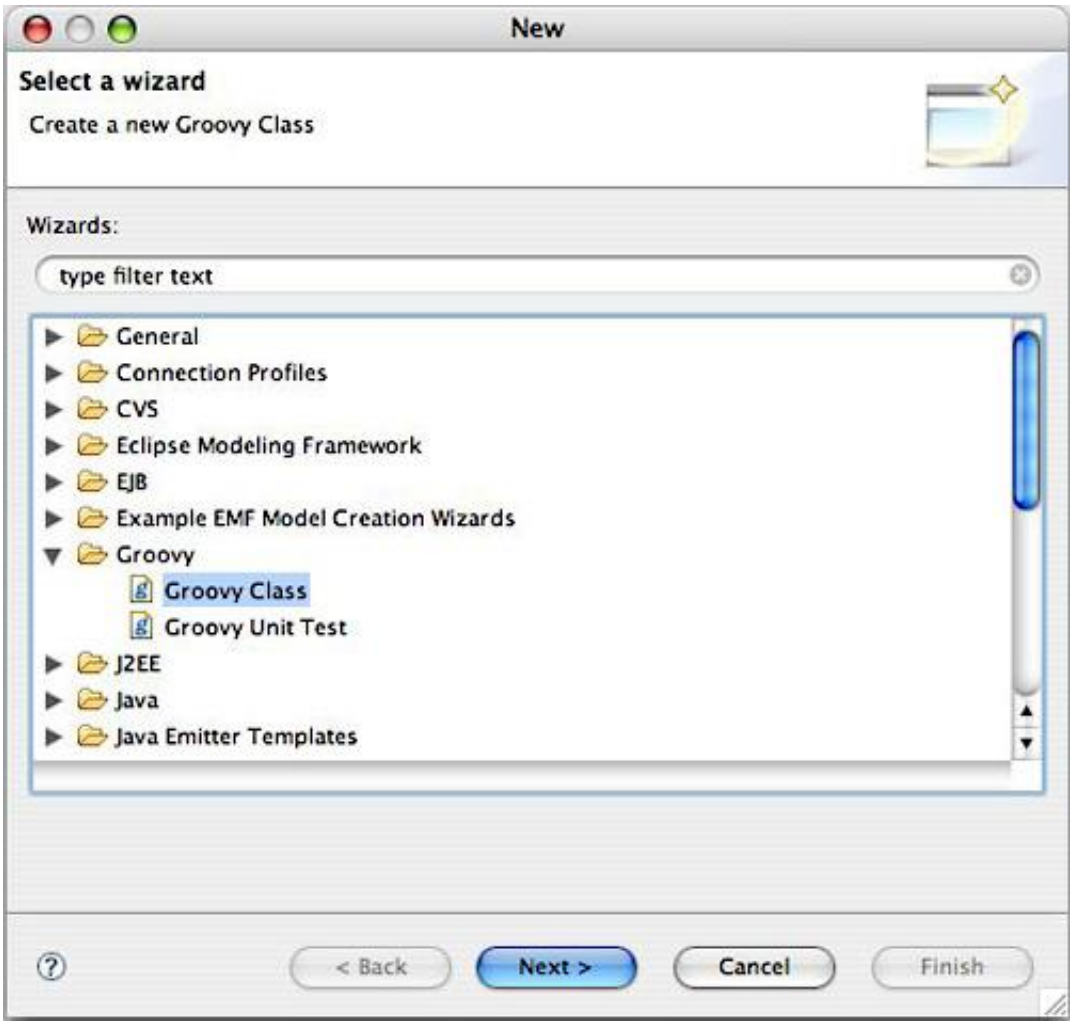
图 7. 通过 New 菜单创建 Groovy 类



给类命名

在这里，找到 Groovy 文件夹，并选择 **Groovy Class** — 应该会看到一个对话框，如图 8 所示。

图 8.选择创建 Groovy 类



单击 **Next** 按钮，系统将要求您提供类的名称。输入 `HelloWorld`。

现在可以将 `HelloWorld` Groovy 类保留在默认包内，如图 9 所示。

图 9. 现在不必考虑包的问题！

虽然步骤看起来很多，但这与创建标准的 Java 类并没有什么区别。

Hello World! — 用 Groovy 编写的 Java 程序

单击 **Finish** 按钮，应该会看到如下所示的代码段：

```
class HelloWorld {
 static void main(args) {

 }
}
```

这看起来同前面的 Java `HelloWorld` 示例惊人地相似。但是请注意，它不包含 `public` 修改符。而且，如果仔细查看 `main` 方法的参数，会注意到它没有类型。

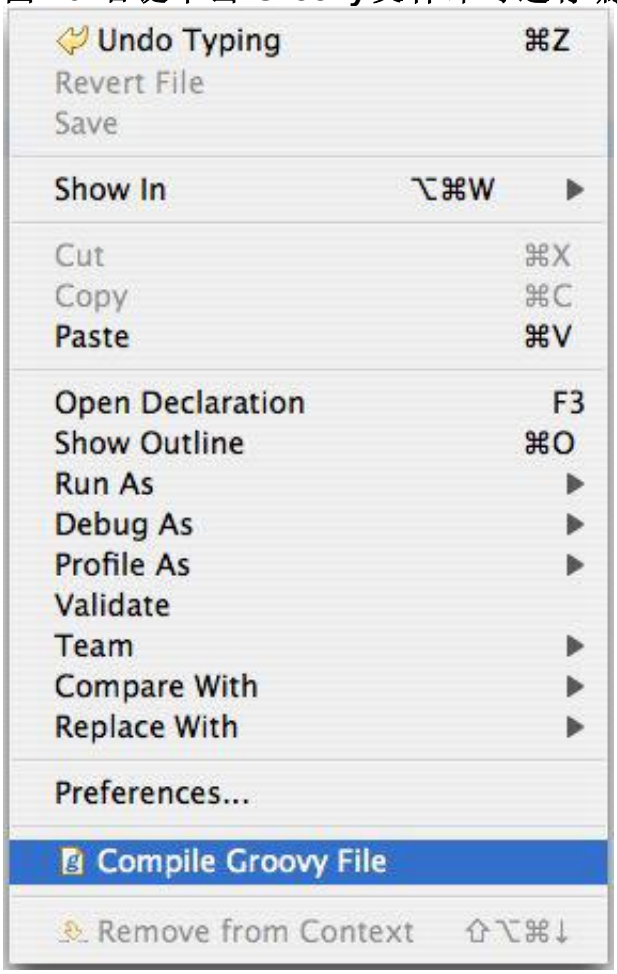
编译程序

现在在 `main` 方法内加入 `println "Hello World"`，完成后的代码看起来如下所示：

```
class HelloWorld {
 static void main(args) {
   println "Hello World"
 }
}
```

在源代码编辑器中应该能够右键单击，并选择 **Compile Groovy File** 选项，如图 10 所示。

图 10. 右键单击 Groovy 文件即可进行编译



运行程序

接下来，再次右键单击文件，选择 **Run As** 选项，然后选择 **Groovy** 选项。在 Eclipse 控制台中应该会看到输出的 “Hello World”，如图 11 所示。

图 11. 输出的 Hello World

学到了什么？

OK，那么这是一种突出重点的取巧方式。Groovy *实际上就是* Java。其语法不同 — 多数情况下会短一些 — 但 Groovy 代码 100% 符合 Java 字节码标准。下一节将进一步介绍这两种语言的交叉。

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

Groovy 变身为 Java

前面已经看到 Groovy 与 Java 代码实际上可以互换的第一个证据。这一节将进一步证明这点，继续使用 Groovy 构建的 `HelloWorld` 类。

Hello, Java!

为了使您确信 Groovy 就是 Java，现在在 `HelloWorld` 类声明和方法声明前面加上 `public` 修改符，如下所示：

```
public class HelloWorld {
 public static void main(args) {
  println "Hello World"
 }
}
```

还不确信？

这个代码运行起来同前面的代码完全一样。但是，如果仍不确信，还可以在 `args` 参数前加上 `String[]`：

```
public class HelloWorld {
 public static void main(String[]args) {
  println "Hello World"
 }
}
```

现在还没完

现在，还可以将 `println` 替换为 `System.out.println` — 而且不要忘记加上括号。

```
public class HelloWorld {
 public static void main(String[] args) {
  System.out.println("Hello World")
  }
}
```

现在的代码与前面用 Java 编写的 Hello World 示例完全相同，但是哪个示例更容易编写呢？

请注意，原来的基于 Groovy 的 `HelloWorld` 类没有任何 `public` 修改符，没有任何类型（没有 `String[]`），而且提供了没有括号的 `println` 快捷方式。

Hello, Groovy!

如果喜欢，可以将这个过程完全翻转过来，回到基于 Java 的 Hello World 示例，删除文件里的*所有内容*，只保留 `System.out` 行，然后在这行删除 `System.out` 和括号。最后只剩下：

```
println "Hello World"
```

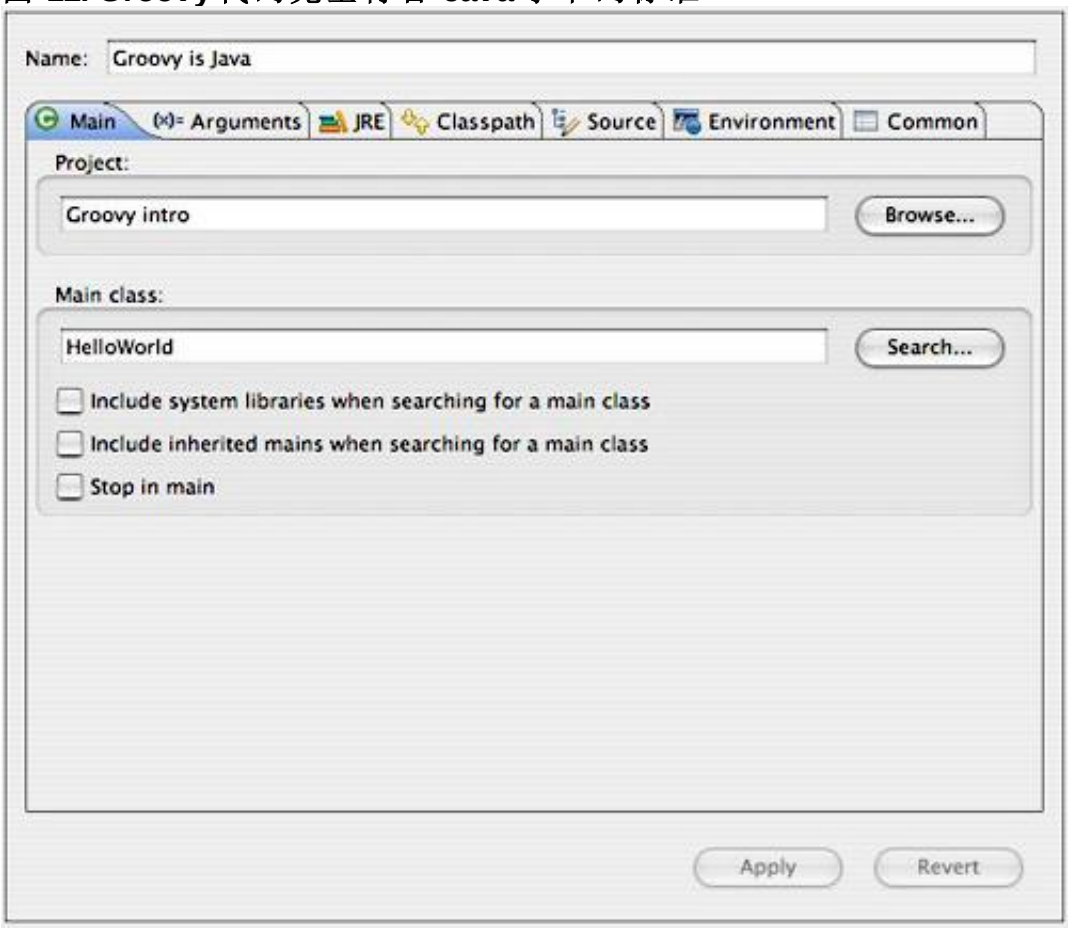
现在，哪个程序更容易编写呢？

运行程序！

Groovy 代码完全符合 Java 字节码标准，这个练习证明了这一点。在 Eclipse 中，选择 **Run** 菜单选项 **Open Run Dialog...**。选择一个新的 **Java Application** 配置。确保项目是您的 Groovy 项目。对于 `Main` 类，单击 **Search** 按钮，找到 `HelloWorld` 类。请注意，单词 *class* 表明 Eclipse Groovy 插件已经将 .groovy 文件编译为 .class 文件。

在图 12 中可以看到整个这个过程 — 如果以前在 Eclipse 中运行过 Java 类，那么您应该对这个过程很熟悉。

图 12. Groovy 代码完全符合 Java 字节码标准



单击 **Run** 按钮，看到什么了？实际上，“Hello World!” 从未像现在这样能够说明问题。

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

Groovy 是没有类型的 Java 代码

很可能将 Groovy 当成是没有规则的 Java 代码。但实际上，Groovy 只是规则少一些。这一节的重点是使用 Groovy 编写 Java 应用程序时可以不用考虑的一个 Java 编程的具体方面：类型定义。

为什么要有类型定义？

在 Java 中，如果要声明一个 `String` 变量，则必须输入：

```
String value = "Hello World";
```

但是，如果仔细想想，就会看出，等号右侧的字符已经表明 `value` 的类型是 `String`。所以，Groovy 允许省略 `value` 前面的 `String` 类型变量，并用 `def` 代替。

```
def value = "Hello World"
```

实际上，Groovy 会根据对象的值来判断它的类型。

运行程序！

将 HelloWorld.groovy 文件中的代码编辑成下面这样：

```
String message = "Hello World"
println message
```

运行这段代码，应该会在控制台上看到与前面一样的 “Hello World”。现在，将变量类型 `String` 替换为 `def` 并重新运行代码。是不是注意到了相同的结果？

除了输出 `message` 的值，还可以用以下调用输出它的类型：

```
def message = "Hello World"
println message.class
```

输出 “`class java.lang.String`” 应该是目前为止很受欢迎的一项变化！如果想知道到底发生了什么，那么可以告诉您：Groovy 推断出 `message` 一定是 `String` 类型的，因为它的值是用双引号括起来的。

类型推断的更多内容

您可能听说过，在 Groovy 中 “一切都是对象” — 但对于类型来说这句话意味着什么呢？让我们看看如果将前面示例中的 `String` 替换为数字会怎么样，如下所示：

```
def message = 12
println message.class
```

`message` 变量的数字值看起来像是 Java 的原生类型 `int`。但是，运行这个代码就可以看出，

Groovy 将它作为 Integer。这是因为在 Groovy 中 “一切都是对象”。

Java 中的所有对象都扩展自 java.lang.Object，这对 Groovy 来说非常方便。即使在最糟的情况下，Groovy 运行时不能确定变量的类型，它只需将变量当成 Object，问题就解决了。

继续使用这段代码。将 message 改成自己喜欢的任意类型：Groovy 会在运行时尽其所能推断出这个变量的类型。

无类型有什么意义

那么，Groovy 缺少类型意味着所需的输入更少。不可否认，将 String 替换成 def 并没有真正节约多少打字工作 — 三个字母并不值得如何夸耀！但是在更高的层次上看，在编写大量不仅仅包含变量声明的代码的时候，没有类型确实减少了许多打字工作。更重要的是，这意味着要阅读的代码要少得多。最后，Groovy 缺少类型能够带来更高的灵活性 — 不需要接口或抽象类。

所以，只需要使用 def 关键字就能在方法中声明一个独立变量，不需要将 def 关键字作为方法声明中的参数。在 for 循环声明中也不需要它，这意味着不用编写 (int x = 0; x < 5; x++)，相反，可以省略 int，保留空白。

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

通过 Groovy 进行循环

同大多数脚本语言一样，Groovy 经常被宣传为*生产力更高* 的 Java 语言替代品。您已经看到了 Groovy 缺少类型能够如何减少打字工作。在这一节，将创建并试用一个 `repeat` 函数。在这个过程中，将进一步探索 Groovy 提高效率的方式。

更好、更短的循环

下面这种方法可以更好地感受 Groovy 缺乏类型的好处：首先，用与创建 `HelloWorld` 相同的方式创建一个 Groovy 类，将这个类称为 `MethodMadness`，并删除自动生成的类体：将要定义一个独立的 `repeat` 函数。现在在控制台中输入以下代码：

```
def repeat(val){
 for(i = 0; i < 5; i++){
  println val
 }
}
```

起初，从 Java 的角度来看，这个小函数看起来可能有些怪（实际上，它很像 JavaScript）。但它就是 Java 代码，只不过是用 Groovy 的样式编写的。

深入方法

`repeat` 函数接受一个变量 `val`。请注意参数不需要 `def`。方法体本质上就是一个 `for` 循环。

调用这个函数。

```
repeat("hello
  world")
```

会输出 “hello world” 五次。请注意，`for` 循环中省略了 `int`。没有变量类型的 `for` 循环要比标准的 Java 代码短些。现在看看如果在代码里加入范围会出现什么情况。

Groovy 中的范围

范围 是一系列的值。例如 “`0..4`” 表明*包含* 整数 0、1、2、3、4。Groovy 还支持排除范围，“`0..<4`” 表示 0、1、2、3。还可以创建字符范围：“`a..e`” 相当于 a、b、c、d、e。“`a..<e`” 包括小于 *e* 的所有值。

循环范围

范围为循环带来了很大的方便。例如，前面从 0 递增到 4 的 `for` 循环如下所示：

```
for(i = 0; i < 5; i++)
```

范围可以将这个 `for` 循环变得更简洁，更易阅读：

```
def repeat(val){
 for(i in 0..5){
  println val
 }
}
```

设置范围

如果运行这个示例，可能会注意到一个小问题："Hello World" 输出了六次而不是五次。这个问题有三种解决方法：

- 将包含的范围限制到 4：

```
for(i in 0..4)
```

- 从 1 而不是 0 开始：

```
def repeat(val){
 for(i in 1..5){
  println val
 }
}
```

- 将范围由包含改为排除：

```
def repeat(val){
 for(i in 0..<5){
  println val
 }
}
```

不论采用哪种方法，都会得到原来的效果 — 输出 "Hello World" 五次。

默认参数值

现在已经成功地使用 Groovy 的范围表达式缩短了 `repeat` 函数。但这个函数依然有些限制。如果想重复 "Hello World" 八次该怎么办？如果想对不同的值重复不同次数 — 比如 "Hello World" 重复八次，"Goodbye Sunshine" 重复两次，这时该怎么办？

每次调用 `repeat` 时都要指定需要的重复次数的做法已经过时了，特别是在已经适应了默认行为（重复五次）的时候。

Groovy 支持*默认参数值*，可以在函数或方法的正式定义中指定参数的默认值。调用函数的程序可以选择省略参数，使用默认值。

更加复杂的参数值

使用前面的 `repeat` 函数时，如果希望调用程序能够指定重复值，可以像下面这样编码：

```
def repeat(val, repeat=5){
 for(i in 0..<repeat){
  println val
 }
}
```

像下面这样调用该函数：

```
repeat("Hello World", 2)
repeat("Goodbye sunshine", 4)
```

```
repeat("foo")
```

结果会输出 “Hello World” 两次，“Goodbye sunshine” 四次，“foo” 五次（默认次数）。

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

Groovy 集合

在 Groovy 提供的所有方便的快捷方式和功能中，最有帮助的一个可能就是*内置的*集合。回想一下在 Java 编程中是如何使用集合的 — 导入 `java.util` 类，初始化集合，将项加入集合。这三个步骤都会增加不少代码。

而 Groovy 可以直接在语言内使用集合。在 Groovy 中，不需要导入专门的类，也不需要初始化对象。集合是语言本身的本地成员。Groovy 也使集合（或者列表）的操作变得非常容易，为增加和删除项提供了直观的帮助。

可以将范围当作集合

在前一节学习了如何用 Groovy 的范围将循环变得更容易。范围表达式 “`0..4`” 代表数字的*集合* — 0、1、2、3 和 4。为了验证这一点，请创建一个新类，将其命名为 `Ranger`。保留类定义和 `main` 方法定义。但是这次添加以下代码：

```
def range = 0..4
println range.class
assert range instanceof List
```

请注意，`assert` 命令用来证明范围是 `java.util.List` 的实例。接着运行这个代码，证实该范围现在是类型 `List` 的集合。

丰富的支持

Groovy 的集合支持相当丰富，而且美妙之处就在于，在 Groovy 的魔法背后，一切都是标准的 Java 对象。每个 Groovy 集合都是 `java.util.Collection` 或 `java.util.Map` 的实例。

前面提到过，Groovy 的语法提供了本地列表和映射。例如，请将以下两行代码添加到 `Ranger` 类中：

```
def coll = ["Groovy", "Java", "Ruby"]
assert  coll instanceof Collection
assert coll instanceof ArrayList
```

你将会注意到，`coll` 对象看起来很像 Java 语言中的数组。实际上，它是一个 `Collection`。要在普通的 Java 代码中得到集合的相同实例，必须执行以下操作：

```
Collection<String> coll = new ArrayList<String>();
coll.add("Groovy");
coll.add("Java");
coll.add("Ruby");
```

在 Java 代码中，必须使用 `add()` 方法向 `ArrayList` 实例添加项。

添加项

Groovy 提供了许多方法可以将项添加到列表 — 可以使用 `add()` 方法（因为底层的集合是一个

普通的 ArrayList 类型），但是还有许多快捷方式可以使用。

例如，下面的每一行代码都会向底层集合加入一些项：

```
coll.add("Python")
coll << "Smalltalk"
coll[5] = "Perl"
```

请注意，Groovy 支持操作符重载 — << 操作符被重载，以支持向集合添加项。还可以通过位置参数直接添加项。在这个示例中，由于集合中只有四个项，所以 [5] 操作符将 “Perl” 放在最后。请自行输出这个集合并查看效果。

检索非常轻松

如果需要从集合中得到某个特定项，可以通过像上面那样的位置参数获取项。例如，如果想得到第二个项 “Java”，可以编写下面这样的代码（请记住集合和数组都是从 0 开始）：

```
assert coll[1] == "Java"
```

Groovy 还允许在集合中增加或去掉集合，如下所示：

```
def numbers = [1,2,3,4]
assert numbers + 5 == [1,2,3,4,5]
assert numbers - [2,3] == [1,4]
```

请注意，在上面的代码中， 实际上创建了*新的* 集合实例，由最后一行可以看出。

魔法方法

Groovy 还为集合添加了其他一些方便的功能。例如，可以在集合实例上调用特殊的方法，如下所示：

```
def numbers = [1,2,3,4]
assert numbers.join(",") == "1,2,3,4"
assert [1,2,3,4,3].count(3) == 2
```

join() 和 count() 只是在任何项列表上都可以调用的众多方便方法中的两个。*分布操作符*（*spread operator*） 是个特别方便的工具，使用这个工具不用在集合上迭代，就能够调用集合的每个项上的方法。

假设有一个 String 列表，现在想将列表中的项目全部变成大写，可以编写以下代码：

```
assert ["JAVA", "GROOVY"] ==
  ["Java", "Groovy"]*.toUpperCase()
```

请注意 *. 标记。对于以上列表中的每个值，都会调用 toUpperCase()，生成的集合中每个 String 实例都是大写的。

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

Groovy 映射

除了丰富的列表处理功能，Groovy 还提供了坚固的映射机制。同列表一样，映射也是本地数据结构。而且 Groovy 中的任何映射机制在幕后都是 `java.util.Map` 的实例。

Java 语言中的映射

Java 语言中的映射是名称-值对的集合。所以，要用 Java 代码创建典型的映射，必须像下面这样操作：

```
Map<String, String>map = new HashMap<String, String>();
map.put("name", "Andy");
map.put("VPN-#","45");
```

一个 `HashMap` 实例容纳两个名称-值对，每一个都是 `String` 的实例。

通过 Groovy 进行映射

Groovy 使得处理映射的操作像处理列表一样简单 — 例如，可以用 Groovy 将上面的 Java 映射写成

```
def hash = [name:"Andy", "VPN-#":45]
```

请注意，Groovy 映射中的键不必是 `String`。在这个示例中，`name` 看起来像一个变量，但是在幕后，Groovy 会将它变成 `String`。

全都是 Java

接下来创建一个新类 `Mapper` 并加入上面的代码。然后添加以下代码，以证实底层的代码是真正的 Java 代码：

```
assert hash.getClass() == java.util.LinkedHashMap
```

可以看到 Groovy 使用了 Java 的 `LinkedHashMap` 类型，这意味着可以使用标准的 Java 一样语句对 `hash` 中的项执行 `put` 和 `get` 操作。

```
hash.put("id", 23)
assert hash.get("name") == "Andy"
```

有 groovy 特色的映射

现在您已经看到，Groovy 给任何语句都施加了魔法，所以可以用 . 符号将项放入映射中。如果想将新的名称-值对加入映射（例如 `dob` 和 “01/29/76”），可以像下面这样操作：

```
hash.dob = "01/29/76"
```

. 符号还可以用来获取项。例如，使用以下方法可以获取 dob 的值：

```
assert hash.dob == "01/29/76"
```

当然 . 要比调用 get() 方法更具 Groovy 特色。

位置映射

还可以使用假的位置语法将项放入映射，或者从映射获取项目，如下所示：

```
assert hash["name"] == "Andy"
hash["gender"] = "male"
assert hash.gender == "male"
assert hash["gender"] == "male"
```

但是，请注意，在使用 [] 语法从映射获取项时，必须将项作为 String 引用。

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

Groovy 中的闭包

现在，闭包是 Java 世界的一个重大主题，对于是否会在 Java 7 中包含闭包仍然存在热烈的争论。有些人会问：既然 Groovy 中已经存在闭包，为什么 Java 语言中还需要闭包？这一节将学习 Groovy 中的闭包。如果没有意外，在闭包成为 Java 语法的正式部分之后，这里学到的内容将给您带来方便。

不再需要更多迭代

虽然在前几节编写了不少集合代码，但还没有实际地在集合上迭代。当然，您知道 Groovy 就是 Java，所以如果愿意，那么总是能够得到 Java 的 `Iterator` 实例，用它在集合上迭代，就像下面这样：

```
def acoll = ["Groovy", "Java", "Ruby"]

for(Iterator iter = acoll.iterator(); iter.hasNext();){
 println iter.next()
}
```

实际上在 `for` 循环中并不需要类型声明，因为 Groovy 已经将迭代转变为任何集合的直接成员。在这个示例中，不必获取 `Iterator` 实例并直接操纵它，可以直接在集合上迭代。而且，通常放在循环构造内的行为（例如 `for` 循环体中 `println`）接下来要放在闭包内。在深入之前，先看看如何执行这步操作。

能否看见闭包？

对于上面的代码，可以用更简洁的方式对集合进行迭代，如下所示：

```
def acoll = ["Groovy", "Java", "Ruby"]

acoll.each{
 println it
}
```

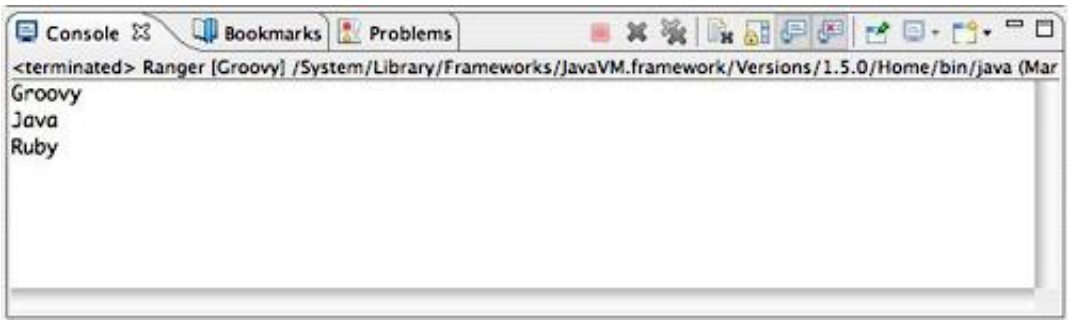
请注意，`each` 直接在 `acoll` 实例内调用，而 `acoll` 实例的类型是 `ArrayList`。在 `each` 调用之后，引入了一种新的语法 — `{`，然后是一些代码，然后是 `}`。由 `{}` 包围起来的代码块就是闭包。

执行代码

闭包是可执行的代码块。它们不需要名称，可以在定义之后执行。所以，在上面的示例中，包含输出 `it`（后面将简单解释 `it`）的行为的无名闭包将会在 `acoll` 集合类型中的每个值上被调用。

在较高层面上，`{}` 中的代码会执行三次，从而生成如图 13 所示的输出。

图 13. 迭代从未像现在这样容易



闭包中的 it 变量是一个关键字，指向被调用的外部集合的每个值 — 它是默认值，可以用传递给闭包的参数覆盖它。下面的代码执行同样的操作，但使用自己的项变量：

```
def acoll = ["Groovy", "Java", "Ruby"]

acoll.each{ value ->
 println value
}
```

在这个示例中，用 value 代替了 Groovy 的默认 it。

迭代无处不在

闭包在 Groovy 中频繁出现，但是，通常用于在一系列值上迭代的时候。请记住，一系列值可以用多种方式表示，不仅可以用列表表示 — 例如，可以在映射、String、JDBC Rowset、File 的行上迭代，等等。

如果想在前面一节 “Groovy 中的映射” 中的 hash 对象上迭代，可以编写以下代码：

```
def hash = [name:"Andy", "VPN-#":45]
hash.each{ key, value ->
 println "${key} : ${value}"
}
```

请注意，闭包还允许使用多个参数 — 在这个示例中，上面的代码包含两个参数（key 和 value）。

使用 Java 代码迭代

以下是使用典型的 Java 构造如何进行同样的迭代：

```
Map<String, String>map = new HashMap<String, String>();
map.put("name", "Andy");
map.put("VPN-#","45");


for(Iterator iter = map.entrySet().iterator(); iter.hasNext();){
 Map.Entry entry = (Map.Entry)iter.next();
 System.out.println(entry.getKey() + " : " + entry.getValue());
}
```

上面的代码比 Groovy 的代码长得多，是不是？如果要处理大量集合，那么显然用 Groovy 处理会更方便。

迭代总结

请记住，凡是集合或一系列的内容，都可以使用下面这样的代码进行迭代。

```
"ITERATION".each{
 println it.toLowerCase()
}
```

闭包的更多使用方式

虽然在迭代上使用闭包的机会最多，但闭包确实还有其他用途。因为闭包是一个代码块，所以能够作为参数进行传递（Groovy 中的函数或方法不能这样做）。闭包在调用的时候才会执行这一事实（不是在定义的时候）使得它们在某些场合上特别有用。

例如，通过 Eclipse 创建一个 `ClosureExample` 对象，并保持它提供的默认类语法。在生成的 `main()` 方法中，添加以下代码：

```
def excite = { word ->
 return "${word}!!"
}
```

这段代码是名为 `excite` 的闭包。这个闭包接受一个参数（名为 `word`），返回的 `String` 是 `word` 变量加两个感叹号。请注意在 `String` 实例中 *替换* 的用法。在 `String` 中使用 `${value}`语法将告诉 Groovy 替换 `String` 中的某个变量的值。可以将这个语法当成 `return word + "!!"` 的快捷方式。

延迟执行

既然有了闭包，下面就该实际使用它了。可以通过两种方法调用闭包：直接调用或者通过 `call()` 方法调用。

继续使用 `ClosureExample` 类，在闭包定义下面添加以下两行代码：

```
assert "Groovy!!" == excite("Groovy")
assert "Java!!" == excite.call("Java")
```

可以看到，两种调用方式都能工作，但是直接调用的方法更简洁。不要忘记闭包在 Groovy 中也是一类对象 — 既可以作为参数传递，也可以放在以后执行。用普通的 Java 代码可以复制同样的行为，但是不太容易。现在不会感到惊讶了吧？

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

Groovy 中的类

迄今为止，您已经用 Groovy 输出了许多次 “Hello World”，已经操作了集合，用闭包在集合上迭代，也定义了您自己的闭包。做所有这些工作时，甚至还没有讨论那个对 Java 开发人员来说至关重要的概念 — 类。

当然，您已经在这个教程中使用过类了：您编写的最后几个示例就是在不同类的 `main()` 方法中。而且，您已经知道，在 Groovy 中可以像在 Java 代码中一样定义类。惟一的区别是，不需要使用 `public` 修改符，而且还可以省略方法参数的类型。这一节将介绍使用 Groovy 类能够进行的其他所有操作。

Song 类

我们先从用 Groovy 定义一个简单的 JavaBean 形式的类开始，这个类称为 `Song`。

第一步自然是用 Groovy 创建名为 `Song` 的类。这次还要为它创建一个包结构 — 创建一个包名，例如 `org.acme.groovy`。

创建这个类之后，删除 Groovy 插件自动生成的 `main()`。

歌曲有一些属性 — 创作歌曲的艺术家、歌曲名称、风格等等。请将这些属性加入新建的 `Song` 类，如下所示：

```
package org.acme.groovy

class Song {
 def name
 def artist
 def genre
}
```

迄今为止还不错，是不是？对于 Grooovy 的新开发人员来说，还不算太复杂！

Groovy 类就是 Java 类

应该还记得本教程前面说过 Groovy 编译器为用 Groovy 定义的每个类都生成标准的 Java `.class`。还记得如何用 Groovy 创建 `HelloWorld` 类、找到 `.class` 文件并运行它么？也可以用新定义的 `Song` 类完成同样的操作。如果通过 Groovy 的 `groovyc` 编译器编译代码（Eclipse Groovy 插件已经这样做了），就会生成一个 `Song .class` 文件。

这意味着，如果想在另一个 Groovy 类或 Java 类中使用新建的 `Song` 类，则必须导入 它（当然，除非使用 `Song` 的代码与 `Song` 在同一个包内）。

接下来创建一个新类，名为 `SongExample`，将其放在另一个包结构内，假设是 `org.thirdparty.lib`。

现在应该看到如下所示的代码：

```
package org.thirdparty.lib

class SongExample {
 static void main(args) {}
}
```

类的关系

现在是使用 Song 类的时候了。首先导入实例，并将下面的代码添加到 SongExample 的 main() 方法中。

```
package org.thirdparty.lib

import org.acme.groovy.Song

class SongExample {
 static void main(args) {
  def sng = new Song(name:"Le Freak",
    artist:"Chic", genre:"Disco")
 }
}
```

现在 Song 实例创建完成了！但是仔细看看以前定义的 Song 类的初始化代码，是否注意到什么特殊之处？您应该注意到自动生成了构造函数。

类初始化

Groovy 自动提供一个构造函数，构造函数接受一个名称-值对的映射，这些名称-值对与类的属性相对应。这是 Groovy 的一项开箱即用的功能 — 用于类中定义的任何属性，Groovy 允许将存储了大量值的映射传给构造函数。映射的这种用法很有意义，例如，您不用初始化对象的每个属性。

也可以添加下面这样的代码：

```
def sng2 = new Song(name:"Kung Fu Fighting", genre:"Disco")
```

也可以像下面这样直接操纵类的属性：

```
def sng3 = new Song()
sng3.name = "Funkytown"
sng3.artist = "Lipps Inc."
sng3.setGenre("Disco")

assert sng3.getArtist() == "Lipps Inc."
```

从这个代码中明显可以看出，Groovy 不仅创建了一个构造函数，允许传入属性及其值的映射，还可以通过 . 语法间接地访问属性。而且，Groovy 还生成了标准的 setter 和 getter 方法。

在进行属性操纵时，非常有 Groovy 特色的是：总是会调用 setter 和 getter 方法 — 即使直接通过 . 语法访问属性也是如此。

核心的灵活性

Groovy 是一种本质上就很灵活的语言。例如，看看从前面的代码中将 setGenre() 方法调用的

括号删除之后会怎么样，如下所示：

```
sng3.setGenre "Disco"
assert sng3.genre == "Disco"
```

在 Groovy 中，对于接受参数的方法，可以省略括号 — 在某些方面，这样做会让代码更容易阅读。

方法覆盖

迄今为止已经成功地创建了 Song 类的一些实例。但是，它们还没有做什么有趣的事情。可以用以下命令输出一个实例：

```
println sng3
```

在 Java 中这样只会输出所有对象的默认 toString() 实现，也就是类名和它的 hashcode（即 org.acme.groovy.Song@44f787）。下面来看看如何覆盖默认的 toString() 实现，让输出效果更好。

在 Song 类中，添加以下代码：

```
String toString(){
 "${name}, ${artist}, ${genre}"
}
```

根据本教程已经学到的内容，可以省略 toString() 方法上的 public 修改符。仍然需要指定返回类型（String），以便实际地覆盖正确的方法。方法体的定义很简洁 — 但 return 语句在哪？

不需要 return

您可能已经想到：在 Groovy 中可以省略 return 语句。Groovy 默认返回方法的最后一行。所以在这个示例中，返回包含类属性的 String。

重新运行 SongExample 类，应该会看到更有趣的内容。toString() 方法返回一个描述，而不是 hashcode。

特殊访问

Groovy 的自动生成功能对于一些功能来说很方便，但有些时候需要覆盖默认的行为。例如，假设需要覆盖 Song 类中 getGenre() 方法，让返回的 String 全部为大写形式。

提供这个新行为很容易，只要定义 getGenre() 方法即可。可以让方法的声明返回 String，也可以完全省略它（如果愿意）。下面的操作可能是最简单的：

```
def getGenre(){
 genre.toUpperCase()
}
```

同以前一样，这个简单方法省略了返回类型和 return 语句。现在再次运行 SongExample 类。应该会看到一些意外的事情 —— 出现了空指针异常。

空指针安全性

如果您一直在跟随本教程，那么应该已经在 SongExample 类中加入了下面的代码：

```
assert sng3.genre == "Disco"
```

结果在重新运行 SongExample 时出现了断言错误 — 这正是为什么在 Eclipse 控制台上输出了丑陋的红色文字。（很抱歉使用了这么一个糟糕的技巧）

幸运的是，可以轻松地修复这个错误：只要在 SongExample 类中添加以下代码：

```
println sng2.artist.toUpperCase()
```

但是现在控制台上出现了*更多的*红色文本 — 出什么事了？！

可恶的 null

如果回忆一下，就会想起 sng2 实例没有定义 artist 值。所以，在调用 toUpperCase() 方法时就会生成 Nullpointer 异常。

幸运的是， Groovy 通过 ? 操作符提供了一个安全网 — 在方法调用前面添加一个 ? 就相当于在调用前面放了一个条件，可以防止在 null 对象上调用方法。

例如，将 sng2.artist.toUpperCase() 行替换成 sng2.artist?.toUpperCase()。请注意，也可以省略后面的括号。（Groovy 实际上也允许在不带参数的方法上省略括号。不过，如果 Groovy 认为您要访问类的属性而不是方法，那么这样做可能会造成问题。）

重新运行 SongExample 类，您会发现 ? 操作符很有用。在这个示例中，没有出现可恶的异常。现在将下面的代码放在这个类内，再次运行代码。

```
def sng4 = new Song(name:"Thriller", artist:"Michael Jackson")
println sng4
```

就是 Java

您将会注意到，虽然预期可能有异常，但是没有生成异常。即使没有定义 genre，getGenre() 方法也会调用 toUpperCase()。

您还记得 Groovy 就是 Java，对吧？ 所以在 Song 的 toString() 中，引用了 genre 属性本身，所以不会调用 getGenre()。现在更改 toString() 方法以使用 getGenre()，然后再看看程序运行的结果。

```
String toString(){
 "${name}, ${artist}, ${getGenre()}"
}
```

重新运行 SongExample，出现类似的异常。现在，请自己尝试修复这个问题，看看会发生什么。

另一个方便的小操作符

希望您做的修改与我的类似。在下面将会看到，我进一步扩充了 Song 类的 getGenre() 方法，以利用 Groovy 中方便的 ? 操作符。

```
def getGenre(){
 genre?.toUpperCase()
}
```

? 操作符时刻都非常有用，可以极大地减少条件语句。

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

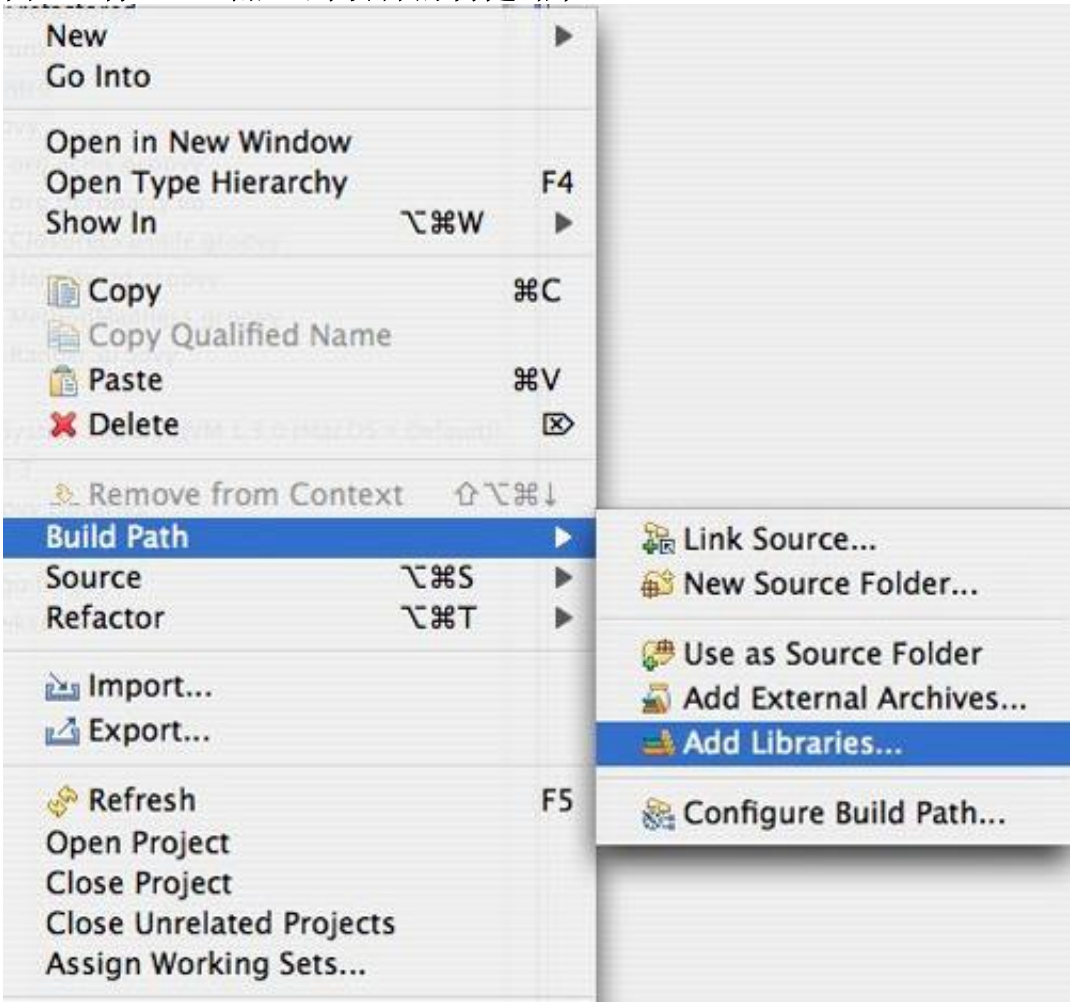
对 Groovy 进行单元测试

本教程一直都强调 Groovy 只是 Java 的一个变体。您已经看到可以用 Groovy 编写并使用标准的 Java 程序。为了最后一次证明这点，在结束本教程之前，我们将通过 JUnit *利用 Java* 对 Song 类进行单元测试。

将 JUnit 加入 Eclipse 项目

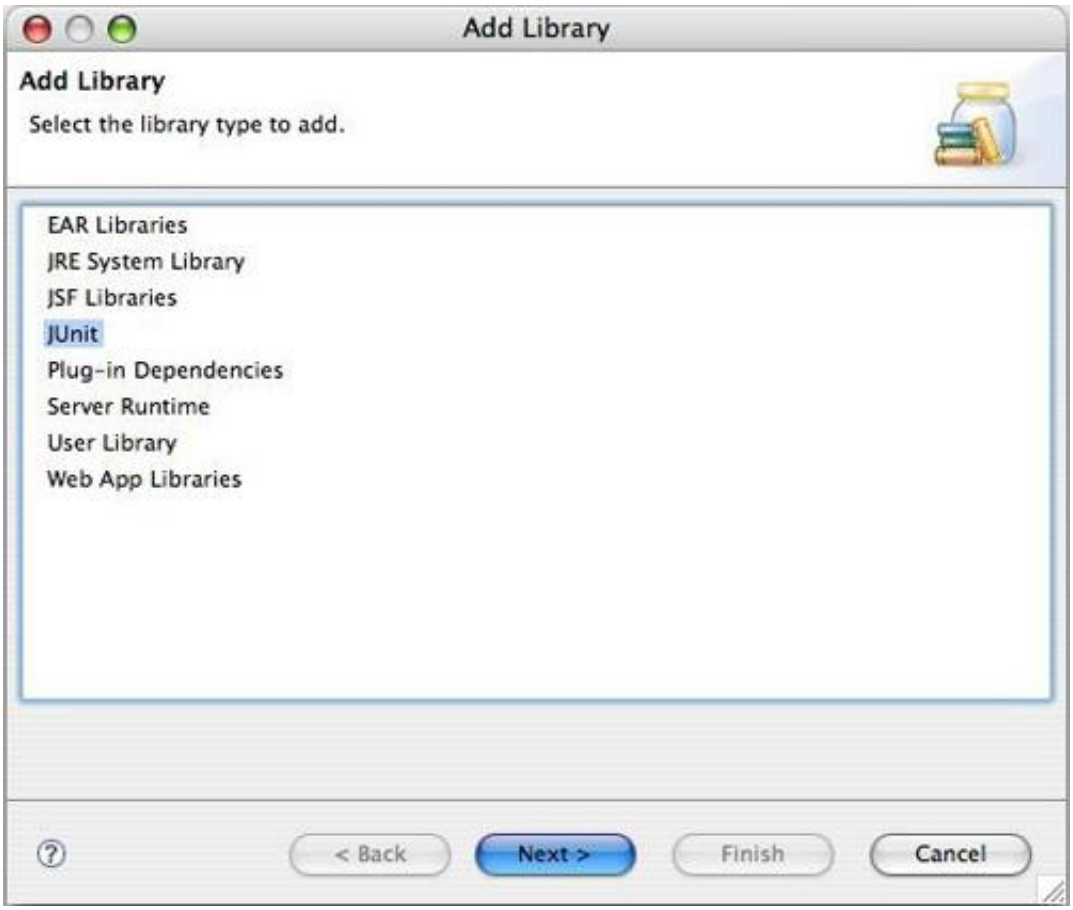
为了跟上本节的示例，需要将 JUnit 加入到 Eclipse 项目中。首先，右键单击项目，选择 **Build Path**，然后选择 **Add Libraries**，如图 14 所示：

图 14. 将 JUnit 加入到项目的构建路径



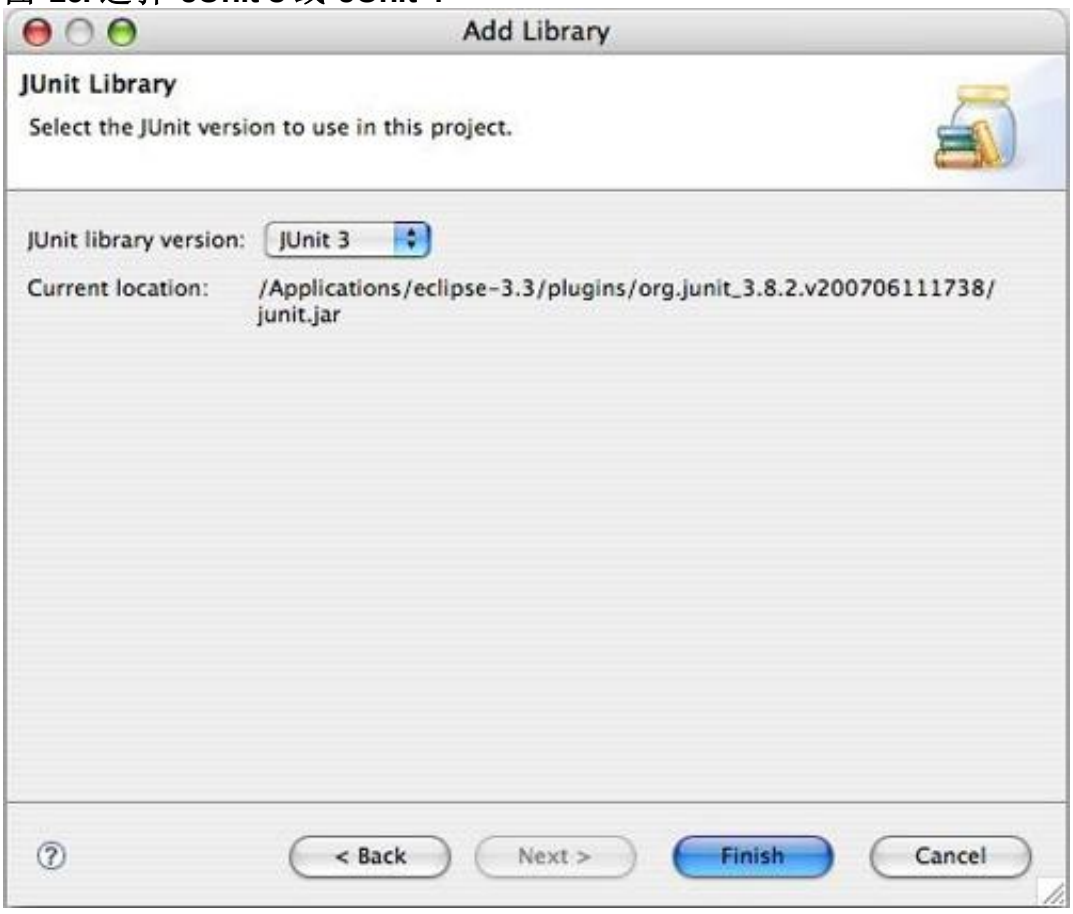
会出现 **Add Library** 对话框，如图 15 所示。

图 15. 从库列表中选择 JUnit



选择 JUnit 并单击 **Next** 按钮。应该会看到如图 16 所示的对话框。选择 **JUnit3** 或 **4** — 具体选择哪项全凭自己决定 — 并单击 **Finish** 按钮。

图 16. 选择 JUnit 3 或 JUnit 4

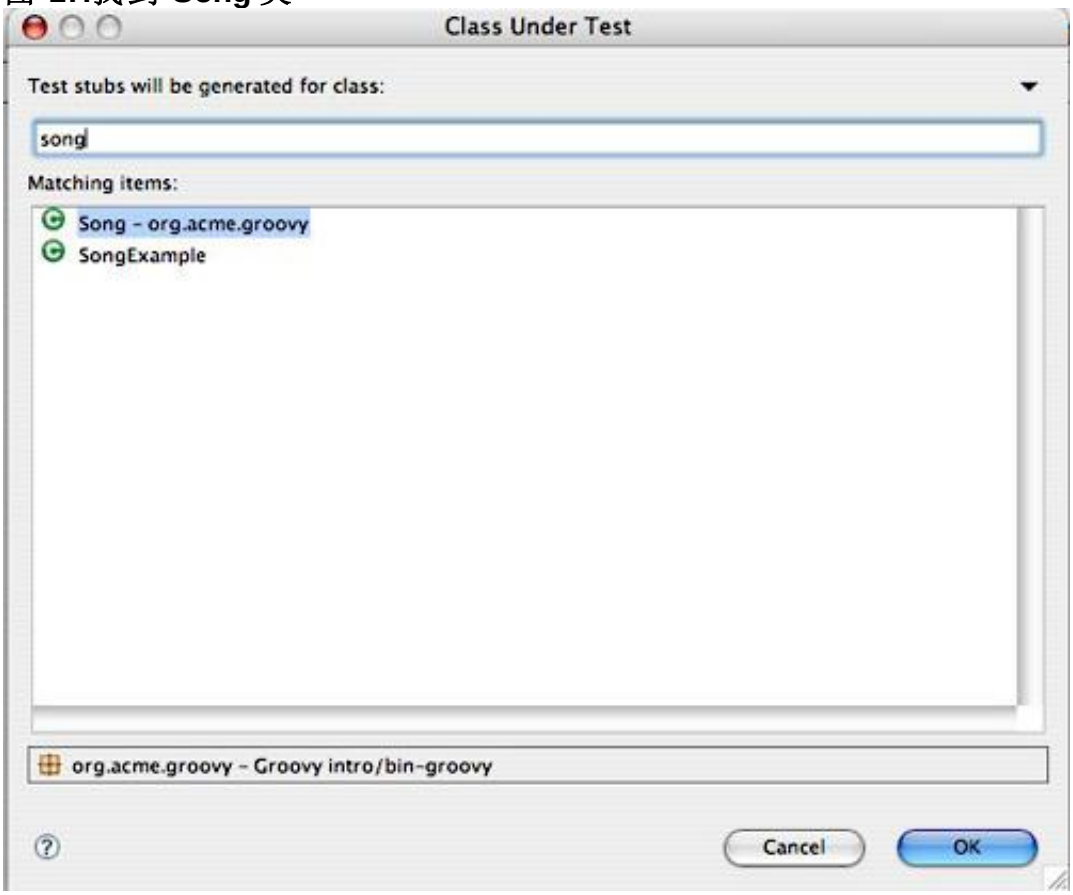


设置新的测试用例

现在在项目的类路径中加入了 JUnit，所以能够编写 JUnit 测试了。请右键单击 java 源文件夹，选择 **New**，然后选择 **JUnit Test Case**。定义一个包，给测试用例命名（例如 `SongTest`），在 Class Under Test 部分，单击 **Browse** 按钮。

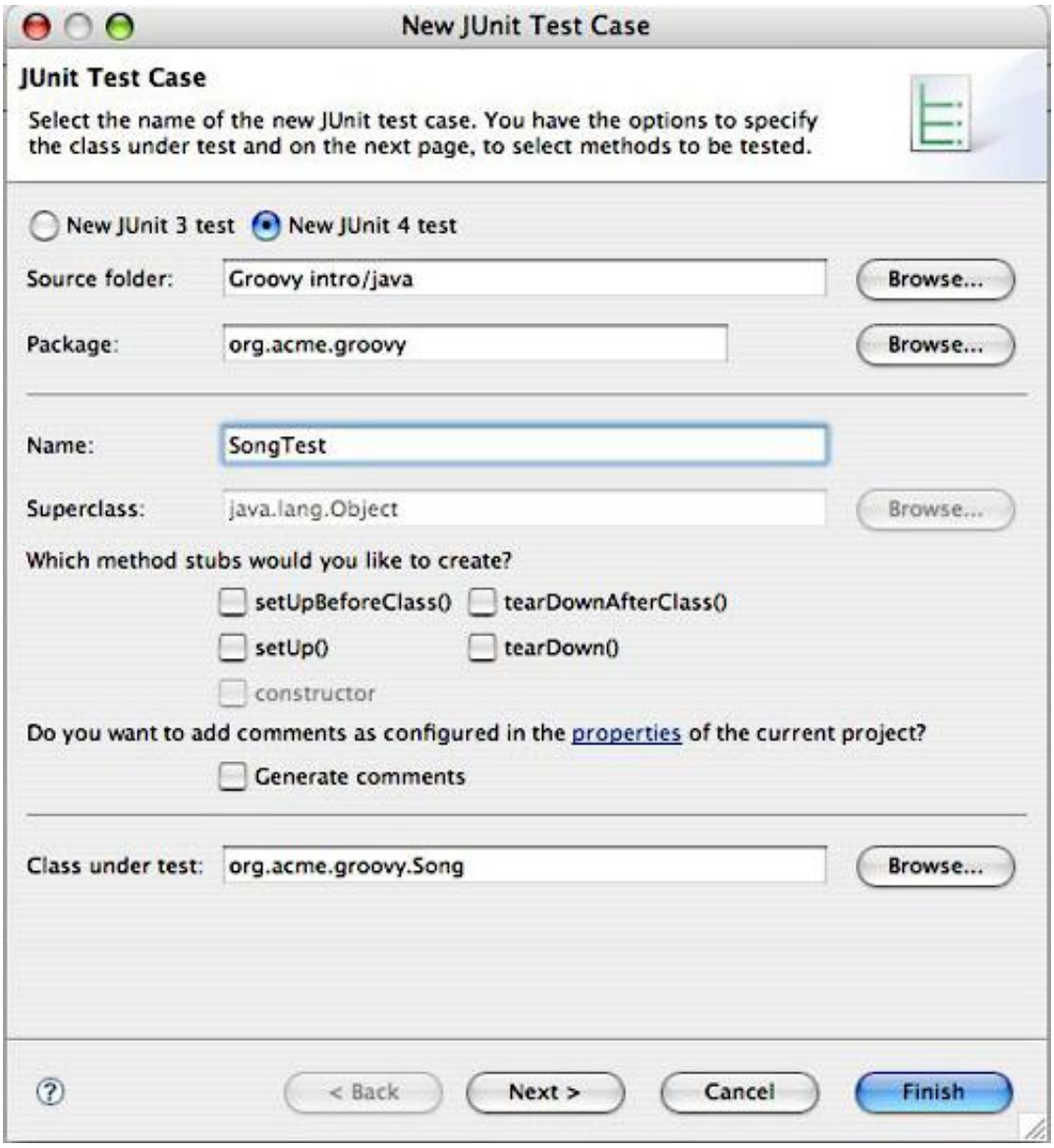
请注意，可以选择用 Groovy 定义的 `Song` 类。图 17 演示了这一步骤：

图 17.找到 Song 类



选择该类并单击 **OK**（应该会看到与图 18 类似的对话框）并在 New JUnit Test Case 对话框中单击 **Finish** 按钮。

图 18. Song 的新测试用例



定义测试方法

我选择使用 JUnit 4；所以我定义了一个名为 `testToString()` 的测试方法，如下所示：

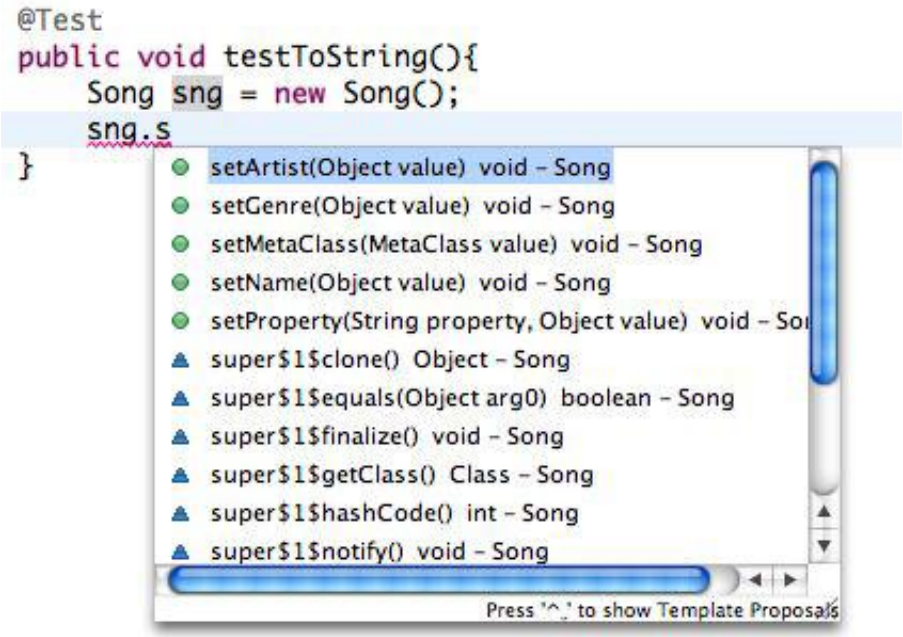
```
package org.acme.groovy;

import org.junit.Test;

public class SongTest {

 @Test
 public void testToString(){}

}
```

测试 toString

显然，需要验证 `toString()` 方法是否没有问题，那么第一步该做什么呢？如果想的是“导入 `Song` 类”，那么想得就太难了 — `Song` 类在同一个包内，所以第一步是创建它的实例。

在创建用于测试的 Song 实例时，请注意不能通过传给构造函数的映射完全初始化 — 而且，如果想自动完成实例的 setter 方法，可以看到每个 setter 接受的是 Object 而不是 String（如图 19 所示）。为什么会这样呢？

图 19. 所有的 setter 和 getter



Groovy 的功劳

如果回忆一下，就会记得我在本教程开始的时候说过：

> 因为 Java 中的每个对象都扩展自 java.lang.Object，所以即使在最坏情况下，Groovy 不能确定变量的类型，Groovy 也能将变量的类型设为 Object然后问题就会迎刃而解。

现在回想一下，在定义 Song 类时，省略了每个属性的类型。Groovy 将自然地将每个属性的类型设为 Object。所以，在标准 Java 代码中使用 Song 类时，看到的 getter 和 setter 的参数类型和返回类型全都是 Object。

修正返回类型

为了增添乐趣，请打开 Groovy Song 类，将 artist 属性改为 String 类型，而不是无类型，如下所示：

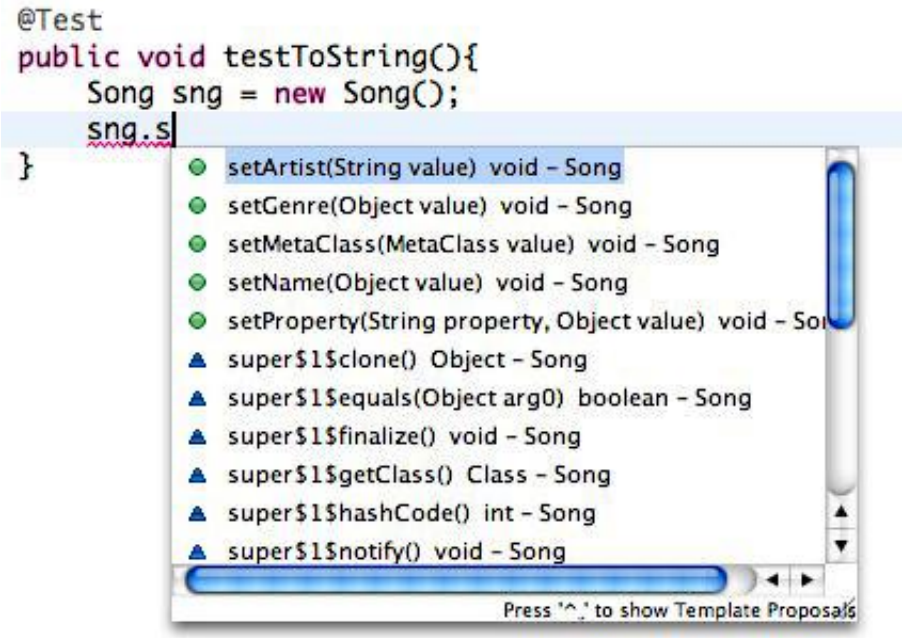
```
package org.acme.groovy

class Song {
 def name
 String artist
 def genre

 String toString(){
  "${name}, ${artist}, ${getGenre()}"
 }
```

```
 def getGenre(){
  genre?.toUpperCase()
 }
}
```

现在，回到 JUnit 测试，在 Song 实例上使用自动完成功能 — 看到了什么？

在图 20 中（以及您自己的代码中，如果一直跟随本教程的话），setArtist() 方法接受一个 String，*而不是* Object。Groovy 再次证明了它就是 Java，而且应用了相同的规则。

图 20. String，而不是 object



始终是普通的 Java

返回来编写测试，另外请注意，默认情况下 Groovy 编译的类属性是私有的，所以不能直接在 Java 中访问它们，必须像下面这样使用 setter：

```
@Test
public void testToString(){
 Song sng = new Song();
 sng.setArtist("Village People");
 sng.setName("Y.M.C.A");
 sng.setGenre("Disco");

 Assert.assertEquals("Y.M.C.A, Village People, DISCO",
   sng.toString());
}
```

编写这个测试用例余下的代码就是小菜一碟了。测试用例很好地演示了这样一点：用 Groovy 所做的一切都可以轻易地在 Java 程序中重用，反之亦然。用 Java 语言执行的一切操作和编写的一切代码，在 Groovy 中也都可以使用。

精通 Groovy

使用 **Groovy** 的简单语法开发 **Java** 应用程序

结束语

如果说您从本教程获得了一个收获的话（除了初次体验 Groovy 编程之外），那么这个收获应该是深入地认识到 Groovy 就是 Java，只是缺少了您过去使用的许多语法规则。Groovy 是没有类型、没有修改符、没有 return、没有 `Iterator`、不需要导入集合的 Java。简而言之，Groovy 就是丢掉了许多包袱的 Java，这些包袱可能会压垮 Java 项目。

但是在幕后，Groovy 就是 Java。

我希望通向精通 Groovy 的这第一段旅程给您带来了快乐。您学习了 Groovy 语法，创建了几个能够体验到 Groovy 的生产力增强功能的类，看到了用 Java 测试 Groovy 类有多容易。还遇到了第一次使用 Groovy 的开发者常见的一些问题，看到了如何在不引起太多麻烦的情况下解决它们。

尽管您可能觉得自己目前对 Groovy 还不是很熟练，但您已经走出了第一步。您可以用目前学到的知识编写自己的第一个 Groovy 程序 — 毕竟，您已经设置好了同时支持 Groovy 和 Java 编程的双重环境！作为有趣的练习，您可以试试用 Gant 设置下一个的自动构建版本，Gant 是基于 Ant 的构建工具，使用 Groovy 来定义构建，而不是使用 XML。当您对 Groovy 更加适应时，可以试着用 Groovy on Grails 构建 Web 应用程序模块 — 顺便说一下，这是下一篇教程的主题。
