Introduction to Spring using Swing

Dependency injection and rich clients

Chad Woolley (thewoolleyman@gmail.com)
Software Developer
Ionami

08 November 2005

This tutorial introduces you to the Spring framework and the concept of dependency injection (also known as Inversion of Control), in the context of writing a simple Java™ Swing GUI application. You will develop a complete, working application from the ground up. You'll also get a taste of the Spring Rich Client Project, a new framework for developing Swing applications under Spring. You'll come away with an appreciation of Spring's versatility and the ways in which it can ease your development tasks.

Before you start

About this tutorial

This tutorial will help you learn the basics of the Spring framework for application development, using Swing, and *dependency injection*, also known as Inversion of Control (IOC). After a brief overview of Spring and dependency injection, the bulk of the tutorial is hands-on, walking you step-by-step through creating a fully functional Swing application -- a to-do list program -- using Spring. The walk-through includes three options for a build environment, with detailed setup instructions for each one. You'll learn the basic use and benefits of the Spring framework in the process. Explanations of other relevant concepts are included along the way. You'll also take a few side trips to learn some good programming practices in the context of the tutorial code.

The last section is a brief and high-level review of the Spring Rich Client (RCP) framework, a subproject of the Spring framework that's working to provide a platform for developing rich-client Swing applications under Spring. The tutorial does not include step-by-step instructions for using the Spring RCP but is a starting point for obtaining the source code and exploring the project on your own.

The tutorial shows each new piece of code that you need to type. The complete source code is also available (see Download). The downloadable source code was created by following along and pasting code directly from the tutorial, so it should be bug-free and identical to what you create as you follow along. Because you'll write the GUI application in the Java programming language, it will run on any platform that Java code runs on.

This tutorial is *not* intended to be a comprehensive overview of the Spring framework or Swing. It does not cover use of Spring to create Web applications or access databases, or more-advanced topics such as Spring's support for aspect-oriented programming (AOP). Explore the tutorial's Resources for books, articles, tutorials, and online references that cover Spring and Swing in greater breadth and detail.

Prerequisites

The audience for this tutorial is Java developers. It is intended to be accessible to multiple experience levels, and even if you are familiar with one topic you might still learn something about another. Some sections discuss related design patterns and development approaches. Feel free to skim past these if you are uninterested or already familiar with them. The following knowledge and skill levels will be helpful:

- Familiarity with the basics of the Java programming language, conventions for Java bean components, and experience with basic application design and development in the Java language.
- Basic knowledge of some build environment -- either Eclipse, Apache Ant, or Apache Maven (see Resources). But even if you are unfamiliar with these environments, you can follow the tutorial's fairly detailed instructions for using them. If you have problems with one, you can try one or both of the other two.
- Basic knowledge of XML syntax -- elements, attributes, and how to maintain a well-formed XML document (see Resources).
- Some basic familiarity with the Swing API, although the tutorial attempts to avoid the use of complex Swing code.

You need a computer with a JDK installed (JDK 1.5 is required to run the Rich Client Project demo in the final section) and an Internet connection for downloading the required tools and libraries.

One of the following is required, or you can use your own build environment or IDE:

- Apache Ant Version 1.6.1 or higher, available at http://ant.apache.org
- Apache Maven Version 1.0.2 or higher, available at http://maven.apache.org
- Eclipse version 3.0, 3.1, or higher, available at http://www.eclipse.org

Options for downloading required dependency JARs

In order to minimize the amount of up-front downloading work required to start the tutorial, both the Ant and Maven build scripts in the tutorial's source code provide support for automatically downloading the required dependency JARs (including Spring itself) from public repositories. This feature is built into Maven, and it is provided in the Ant script by the `get-dependencies` target. To use these features with Eclipse, you must have Ant or Maven installed. You'll find more details in Environment setup.

You can obtain the correct required JAR versions manually if you prefer. Just refer to the `get-dependencies` target in the Ant build.xml file (see Listing 1) to find out which JARs and versions are required, as well as the URLs to download them from.

Overview of Spring and dependency injection

What is the Spring framework?

The Spring framework Web site welcomes visitors with this description: "As the leading full-stack Java/J2EE application framework, Spring delivers significant benefits for many projects, reducing development effort and costs while improving test coverage and quality" (see Resources). This generic statement is an indicator of Spring's scope and size. Simply put, Spring provides many tools and approaches that make it easier to write and test Java and J2EE applications. This tutorial only illustrates basic use of dependency injection, but Spring contains many useful and easy-to-use wrappers for other services and frameworks, as well as advanced features such as support for aspect-oriented programming (AOP) (see Resources).

For more overview and background information on Spring, check out the project's mission statement, *The Spring series* on developerWorks, and other articles in Resources.

What is dependency injection?

Dependency Injection (DI), also referred to as Inversion of Control (IOC), is an approach to software development in which a separate object or framework (such as the Spring framework) is responsible for creating and "injecting" objects into other objects that depend on them. This results in code that is loosely coupled and easy to test and reuse.

Check out Martin Fowler's site for some good descriptions of DI/IOC (see Resources).

Environment setup

Pick your build environment

Before you start coding the sample application, you need to set up an environment to build and run it. If you are experienced enough with Java programming to handle building and running on your own, you can skip to Creating the to-do list: Basic Swing and Spring application setup.

You have three options to choose from for a build environment (see Prerequisites). You can pick one that you're familiar with or try a new one to get some exposure to it. You can use any one or all three:

- Apache Ant: You can run the Ant build script from the command line (which requires that you download and install Ant) or from within an IDE (such as Eclipse).
- Apache Maven: Maven is a popular build environment. To use it, you must download and install it, but all of the required configuration files to build with Maven are included in the tutorial.
- Eclipse: Note that even if you use Eclipse to build and run, you still need to obtain the library JAR dependencies and then define references to these dependencies in your Eclipse classpath. You can use the Ant or Maven build script to auto-download the required dependencies to your local machine, or download them yourself. You'll find more details in Set up Eclipse.

Set up Ant

If you've chosen Ant as your build environment:

1. Install Ant and ensure that the Ant executable is on your path.
2. Type `ant -version` at the command line to verify that Ant is installed correctly. Alternatively, you can use Ant from within your IDE of choice or another tool.
3. After you've configured Ant in your operating system or IDE, create a directory named todo. This directory will contain all your project files.
4. Create the build.xml build script, shown in Listing 1, in the directory's root.

Note: For presentation purposes, some lines in the code listings are split at places where you wouldn't ordinarily split them. In some cases, such as Listing 1, this results in an invalid file that you must fix. In these situations, a NOTE is included inline in the code listing instructing you to delete the note and join the lines together with no spaces. (Ant appears to be smart enough to strip newlines and spaces from URLs, but it's better to make them look nice anyway.)

Listing 1. build.xml

```
<project name="todo" default="default">
  <property name="build.dir" location="build"/>
  <property name="lib.dir" location="lib"/>

  <path id="classpath">
    <pathelement location="${build.dir}"/>
    <fileset dir="${lib.dir}"/>
  </path>

  <target name="clean">
    <delete dir="${build.dir}"/>
  </target>

  <target name="get-dependencies">
    <mkdir dir="${lib.dir}"/>
    <get dest="${lib.dir}/commons-logging-1.0.3.jar"
       usetimestamp="true" ignoreerrors="true"
       src="http://www.ibiblio.org/maven/
NOTE: The URL in the lines above and below this note was split for
readability.  Delete this note and join the URL with no spaces
       commons-logging/jars/commons-logging-1.0.3.jar"/>
    <get dest="${lib.dir}/spring-1.2.3.jar"
       usetimestamp="true" ignoreerrors="true"
       src= "http://www.ibiblio.org/maven/
NOTE: The URL in the lines above and below this note was split for
readability.  Delete this note and join the URL with no spaces
       springframework/jars/spring-1.2.3.jar"/>
  </target>

  <target name="compile">
    <mkdir dir="${build.dir}/classes"/>
    <javac srcdir="src"
           destdir="${build.dir}/classes"
           classpathref="classpath"
           encoding="UTF8"
           debug="on"
           deprecation="on"
    />
    <copy todir="${build.dir}/classes" overwrite="true">
        <fileset dir="src">
            <include name="**/*.xml"/>
        </fileset>
    </copy>
  </target>

  <target name="run" depends="compile">
```

```
    <java classname="todo.ToDo" fork="true">
      <classpath>
        <path refid="classpath"/>
        <pathelement location="${build.dir}/classes"/>
      </classpath>
    </java>
  </target>

  <target name="default" depends="get-dependencies, compile, run"/>
</project>
```

Automatically downloading dependencies via Ant

Now, run the `get-dependencies` target, either by typing `ant get-dependencies` on the command line or by running the target in your IDE or editor. You should see output that looks like this:

```
Buildfile: build.xml

get-dependencies:
    [mkdir] Created dir: D:\projects\todo\lib
      [get] Getting: http://www.ibiblio.org/
          maven/commons-logging/jars/commons-logging-1.0.3.jar
      [get] Getting: http://www.ibiblio.org/maven/
          springframework/jars/spring-1.2.3.jar

BUILD SUCCESSFUL
Total time: 30 seconds
```

You will be instructed to use other targets in the Ant script later in the tutorial. The main ones are `clean`, `compile`, and `run`, which behave like most standard Ant scripts.

Set up Maven

If you've chosen to use Maven as the build environment:

1. Install Maven and ensure that the Maven executable is on your path. Refer to the documentation on the Maven site for more details (see Resources).
2. Type `maven -v` at the command line to show the version and verify that Maven is installed correctly. You can use Maven from within Eclipse via an Eclipse or a Maven plug-in; see the Maven site for details.
3. After you've configured Maven in your operating system or IDE, create a directory named todo. This directory will contain all your project files.
4. Create the project.xml (see Listing 2) and maven.xml (see Listing 3) Maven configuration files in the todo directory's root.

Listing 2. project.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <pomVersion>3</pomVersion>
  <id>todo</id>
  <name>To Do List</name>
  <currentVersion>1.0</currentVersion>
  <organization>
    <name>To Do List Inc.</name>
    <url>http://localhost/notavailable</url>
    <logo>http://maven.apache.org/images/jakarta-logo-blue.gif</logo>
  </organization>
  <inceptionYear>2005</inceptionYear>
  <package>todo</package>
  <logo>http://maven.apache.org/images/maven.jpg</logo>
  <description>A To Do List</description>
  <shortDescription>A To Do List</shortDescription>
  <url/>
  <issueTrackingUrl/>
  <siteAddress/>
  <siteDirectory/>
  <distributionDirectory>/notavailable/</distributionDirectory>
  <repository>
    <connection/>
    <url/>
  </repository>
  <mailingLists/>
  <developers/>
  <dependencies>
    <dependency>
        <groupId>springframework</groupId>
        <artifactId>spring</artifactId>
        <version>1.2.3</version>
    </dependency>
    <dependency>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
        <version>1.0.3</version>
    </dependency>
  </dependencies>
  <build>
    <nagEmailAddress/>
    <sourceDirectory>src</sourceDirectory>
    <unitTestSourceDirectory/>
    <unitTest/>
    <resources>
      <resource>
        <directory>src</directory>
        <includes>
          <include>**/*.xml</include>
        </includes>
      </resource>
    </resources>
  </build>
</project>
```

Listing 3. maven.xml

```
<project default="test"
    xmlns:m="maven"
    xmlns:ant="jelly:ant">

    <goal name="run">
        <ant:path id="run.classpath" >
            <ant:pathelement path="${maven.test.dest}"/>
            <ant:pathelement path="${maven.build.dest}"/>
            <ant:path refid="maven.dependency.classpath"/>
        </ant:path>

        <attainGoal name="java:compile"/>
        <attainGoal name="java:jar-resources"/>

        <ant:java classname="todo.ToDo" fork="true">
            <ant:classpath refid="run.classpath"/>
        </ant:java>
    </goal>

</project>
```

Automatically downloading dependencies via Maven

Now, type `maven` at the command line in the root of the todo directory. The output you see should look something like this:

```
 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0.2

Attempting to download spring-1.2.3.jar.
1787K downloaded
Attempting to download commons-logging-1.0.3.jar.
30K downloaded
build:start:

java:prepare-filesystem:

java:compile:
    [echo] Compiling to D:\projects\todo/target/classes
    [echo] No java source files to compile.

java:jar-resources:

test:prepare-filesystem:

test:test-resources:

test:compile:
    [echo] No test source files to compile.

test:test:
    [echo] No tests to run.
BUILD SUCCESSFUL
Total time: 35 seconds
```

Don't worry about the "No tests to run" messages. You are looking for the dependencies to download successfully. Also, be aware that they will download only once, so you won't get the download message on subsequent Maven runs.

You will be instructed to use specific goals in Maven later in the tutorial. The main ones are `clean` and `java:compile`, which are standard in Maven, and `run`, which is a custom goal defined in maven.xml.

Set up Eclipse

This section applies if you've chosen Eclipse as the build environment for the sample application. You'll use Eclipse to create a project, create new files, change perspectives and views, configure your build path, and so on. If you are completely new to Eclipse, I suggest you visit the Eclipse.org main page first to learn the basics of getting around and building projects in Eclipse (see Resources).

First, go to the Java perspective and create a new Java project in your Eclipse workspace named todo. If you already have the todo directory in your workspace (perhaps if you already set up the project for Ant or Maven), you can use the same directory, even though you might get a warning that the location already exists.

After the project is created, you can open the Navigator view (you can't see .* project files in the Package Explorer view) and then create/edit/replace the relevant Eclipse configuration files, shown in Listings 4 through 7. All paths are relative to the todo project root. Usually you wouldn't directly create or edit these files (you would use the menu options in Eclipse), but they are included here for completeness and to reference if you have problems with your Eclipse project setup.

You can overwrite the .project file, shown in Listing 4, or you can use the one that was generated when you created the project:

Listing 4. .project

```
<?xml version"1.0" encoding="UTF-8"?>
<projectDescription>
  <name>todo</name>
  <comment></comment>
  <projects>
  </projects>
  <buildSpec>
    <buildCommand>
      <name>org.eclipse.jdt.core.javabuilder</name>
      <arguments>
      </arguments>
    </buildCommand>
  </buildSpec>
  <natures>
    <nature>org.eclipse.jdt.core.javanature</nature>
  </natures>
</projectDescription>
```

The .cvsignore file, shown in Listing 5, is optional, but you'll want to use it if you check your project into a CVS repository:

Listing 5. .cvsignore

```
target
build
lib
eclipseclasses
```

Use the version of .classpath in Listing 6 if you want to download your dependencies via Ant or if you have downloaded them yourself and placed them in the lib subdirectory:

Listing 6. .classpath pointing to Ant-downloaded dependencies in lib

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
  <classpathentry kind="src" path="src"/>
  <classpathentry kind="con"
     path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
  <classpathentry kind="lib" path="lib/commons-logging-1.0.3.jar"/>
  <classpathentry kind="lib" path="lib/spring-1.2.3.jar"/>
  <classpathentry kind="output" path="eclipseclasses"/>
</classpath>
```

Use the version of .classpath in Listing 7 if you want to download your dependencies via Maven:

Listing 7. .classpath pointing to Maven-downloaded dependencies in MAVEN_REPO

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
  <classpathentry kind="src" path="src"/>
  <classpathentry kind="con"
     path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
  <classpathentry kind="var"
     path= "MAVEN_REPO/commons-logging/jars/commons-logging-1.0.3.jar"/>
  <classpathentry kind="var"
     path="MAVEN_REPO/springframework/jars/spring-1.2.3.jar"/>
  <classpathentry kind="output" path="eclipseclasses"/>
</classpath>
```

Downloading and defining dependency JARs in the Eclipse classpath

You must download the proper dependency JARs in order to run the sample application under Eclipse. To make this easier, both the Ant and Maven builds described in this section provide a way to download the JARs automatically to a specified location (see Automatically downloading dependencies via Ant and Automatically downloading dependencies via Maven).

To auto-download the dependencies using Ant:

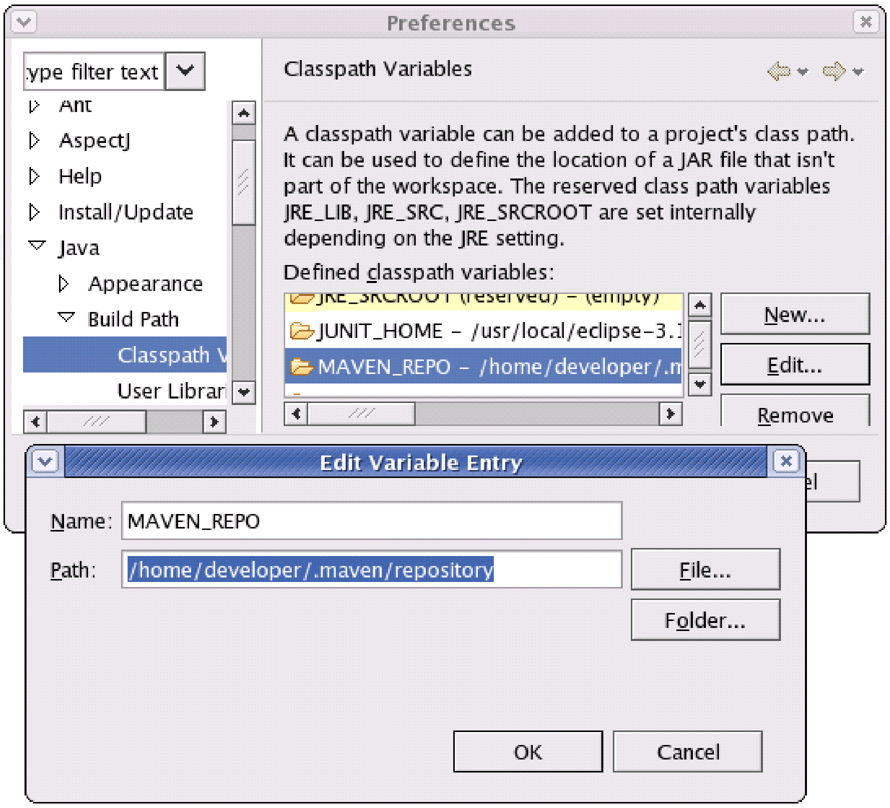
1. Have Ant installed.
2. Create build.xml (see Listing 1).
3. Type `ant get-dependencies` from the root of the project.

If you wish to use Maven to auto-download the dependencies:

1. Have Maven installed.
2. Create project.xml (see Listing 2).
3. Type `ant get-dependencies` from the root of the project.

If you use the Maven approach, you also need to define an Eclipse classpath variable named `MAVEN_REPO` to specify the location of your Maven repository, as shown in Figure 1. (By default, it is in your user home directory.)

Figure 1. Setting an Eclipse classpath variable pointing to MAVEN_REPO



A classpath variable can be added to a project's class path. It can be used to define the location of a JAR file that isn't part of the workspace. The reserved class path variables JRE_LIB, JRE_SRC, JRE_SRCROOT are set internally depending on the JRE setting.

Defined classpath variables:

- JRE_SRCROOT (reserved) - (empty)
- JUNIT_HOME - /usr/local/eclipse-3.1
- MAVEN_REPO - /home/developer/.m

Name: MAVEN_REPO

Path: /home/developer/.maven/repository

Finally, if you don't want to use Ant or Maven, or the auto-download is not working for some reason, you can manually download the required dependencies and place them in the lib folder. In this case, you would use the .classpath example in Listing 6, which references the JARs in the lib folder.

Also, remember to switch back to the Package Explorer view if you are still on the Navigator view. Working with Java code is easier in the Package Explorer view.

Creating the to-do list: Basic Swing and Spring application setup

In this section, you'll create the basic runnable skeleton of the to-do list application, including Swing and Spring files that you'll build on later in the tutorial. You'll perform a simple example of dependency injection. At the end of the section, you'll have a running application.

Creating the MainFrame, Launcher, and ToDo classes

To start the foundation for the Swing application, you need three parts:

- A class that subclasses the Swing `JFrame` class. All Swing applications must have a main outer frame to contain all other components. You'll call this class `MainFrame`.
- A `Launcher` class, responsible for initializing and configuring the Spring framework.
- A class with a `main` method, used to launch the application. You'll name this class `ToDo`.

You could have combined these three separate classes into one or two classes or inner classes, but it's simpler to keep them separate. Separating them has additional advantages in more-complex applications. For example, during testing, you might want to have a specialized `Launcher` class that you can configure and invoke directly from your tests -- perhaps to avoid starting asynchronous tasks that would interfere with testing but are needed during normal application startup.

Listings 8, 9, and 10 show the code for the `MainFrame`, `Launcher`, and `ToDo` classes, respectively. Create a src directory in the project's root. Then create MainFrame.java, Launcher.java, and ToDo.java in the appropriate package structure. (This means that the directory structure under src must match the package name of the class.) Note that MainFrame.java is in the `todo.ui` subpackage, where you'll keep the classes related to the user interface.

Listing 8. src/todo/ui/MainFrame.java

```
package todo.ui;

import java.awt.Dimension;
import java.awt.Frame;
import javax.swing.JFrame;
import javax.swing.WindowConstants;


public class MainFrame extends JFrame {
    public void init() {
        setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        setSize(new Dimension(600, 400));

        setVisible(true);
        setState(Frame.NORMAL);
        show();
    }
}
```

The MainFrame class is a pretty straightforward implementation of a Swing JFrame. The code to configure and show the frame is defined in a `public void init()` method. This method will be the first method in the application invoked by Spring and the entry point to the Swing application. You'll see how to invoke it in the next panel, Creating the Spring app-context.xml bean definition file.

Listing 9. src/todo/Launcher.java

```
package todo;

import
  org.springframework.context.support.ClassPathXmlApplicationContext;


public class Launcher {
    public void launch() {
        String[] contextPaths = new String[] {"todo/app-context.xml"};
        new ClassPathXmlApplicationContext(contextPaths);
    }
}
```

The purpose of the Launcher class is to initialize and launch the Spring framework by creating an ApplicationContext and passing it an array containing the paths to the bean definition file(s) that you'll create in Creating the Spring app-context.xml bean definition file. Spring creates the `MainFrame` class automatically when the framework starts up, because the bean will be defined as a Singleton (see Resources). There are several other types of `ApplicationContext` implementations besides `ClassPathXmlApplicationContext`, but they all serve as a way to provide configuration for a Spring application.

Listing 10. src/todo/ToDo.java

```
package todo;

public class ToDo {
    public static void main(String[] args) {
        Launcher launcher = new Launcher();
        launcher.launch();
    }
}
```

The `ToDo` class in Listing 10 simply has a `main` method that creates a `Launcher` and calls `launch()` on it.

After you type these classes in, make sure they compile by using the appropriate method for your build environment:

- Ant: Change to the root directory of the project that contains build.xml. Type `ant compile` at the command line, or invoke the `compile` target from your IDE.
- Maven: Change to the root directory of the project that contains maven.xml. Type `maven java:compile` at the command line.
- Eclipse: Choose **Build Project**, or ensure you have **Build Automatically** turned on.

You should get a `BUILD SUCCESSFUL` message at the command line or no errors for the project in the Eclipse Problems view. If you have any problems, read the compiler error messages carefully. Specifically, make sure you have the required dependency JARs on your classpath. If you don't have them, go back to Environment setup and read how to download the required dependencies automatically.

Note: From this point forward, you will be expected to compile all new files and changes immediately after you make them and to fix any problems that occur. For the sake of brevity, you won't always be explicitly instructed to compile.

Creating the Spring app-context.xml bean definition file

The heart of a Spring project is the bean definition file (or files). This is an XML file that can have any name. You'll call yours app-context.xml and create it in the `todo` package, where it will be on the classpath (see Listing 11).

Listing 11. src/todo/app-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
          "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="mainFrame" class="todo.ui.MainFrame" init-method="init">
  </bean>
</beans>
```

The root element of the bean definition file is `<beans>`, which contains `<bean>` elements. The `<bean>` element provides several attributes, but for your first bean definition you'll use just three: id, class, and init-method. The `id` attribute defines the bean's name, which is used to retrieve it from Spring. The `class` attribute tells Spring which class to instantiate when it creates the bean. The `init-method` attribute defines a method name on the class that will be automatically invoked after Spring instantiates it.

It is important to know that all beans are Singletons by default, unless you specify the singleton="false" attribute on the bean element. Spring automatically instantiates all Singletons when it is first initialized, unless the `lazy-init="true"` attribute is specified.

This autocreation of Singletons is the reason why the `Launcher` class needs to create only the `ApplicationContext` and isn't required to do anything else. Spring simply creates `MainFrame` as a Singleton and calls the `init()` method on it, which causes it to show itself. Can't wait to try it out? Move on to the next panel, Running the application, for instructions on running your new Spring application.

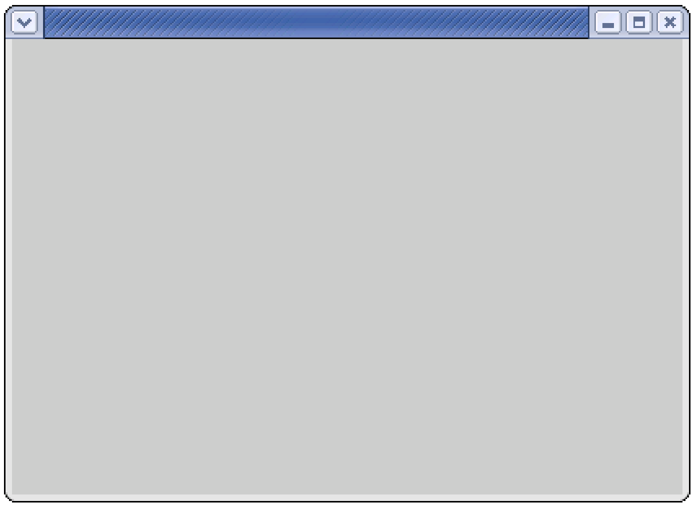
Running the application

To see your application, follow the instructions for your build environment:

- Ant: Change to the root directory of the project that contains build.xml. Type `ant run` or invoke the `run` target from your IDE.
- Maven: Change to the root directory of the project that contains maven.xml. Type `maven run`.
- Eclipse: Run the `ToDo` class as a Java Application (right-click on **ToDo.java** and select **Run As** > **Java Application**).

When you run the application, you should see a blank, gray frame with no title, as shown in Figure 2. If you do, congratulations -- you just ran your first application under Spring!

Figure 2. A blank screen shown by a Swing JFrame, created by Spring



If it doesn't run, look for any exception messages on the console. Spring usually provides straightforward and descriptive error messages and exceptions, so read them carefully. Even if the application runs successfully, you might see some `INFO` messages printed to the console, such as "`INFO: Unable to locate MessageSource with name 'messageSource': using default...`." Don't worry, these are normal.

Note: From this point forward, the instruction "run the application" means that you should run the appropriate command for your build environment.

Defining bean properties

Now that you have a basic bean being run by Spring, you can try defining some properties for the bean in app-context.xml. You'll start by defining a title for the bean. The `setTitle(String title)` accessor method for the frame title is defined on the Frame superclass of your `MainFrame` class, so you can use it. Add the new code in Listing 12 to your definition of the `mainFrame` bean in app-context.xml. Remember, you already created the src/todo/app-context.xml file, so you only need to add the new lines. From now on, if a code listing is for an existing file, the new (boldfaced) lines will be surrounded by existing lines so you know where to put the new lines.

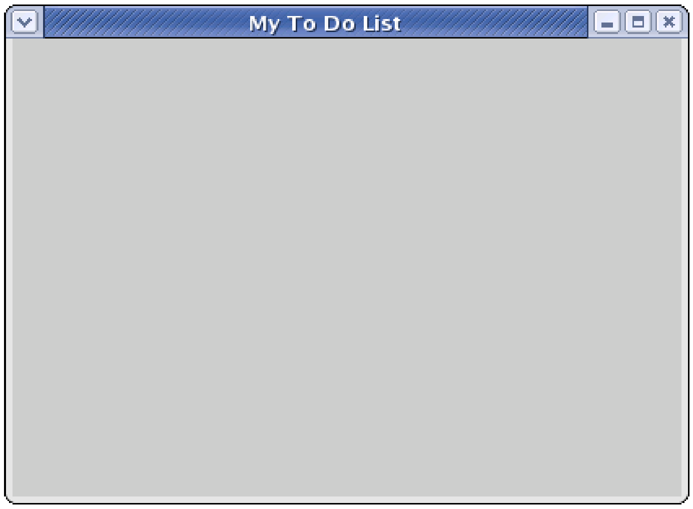
Spring also gives you an easy and powerful way to change simple properties such as a window title without needing to edit the configuration file or even recompile the application. You can define these values in property files that you can keep in the filesystem outside of your application. This capability lets you do handy things like letting each developer on your team have local overrides in a property file on his or her own machine (outside of the application directory tree, so they don't get overwritten on new checkouts). This is great for overriding things like the location and ID/password for their personal development database or the e-mail address to use when the application sends test e-mails. You can also use this technique as a way to do "emergency" property overrides for applications running in production, without needing to rebuild or redeploy them. To learn more, read about BeanFactoryPostProcessors in the Spring documentation.

Listing 12. Defining the bean's title property

```
<bean id="mainFrame" class="todo.ui.MainFrame" init-method="init">
  <property name="title">
    <value>My To Do List</value>
  </property>
</bean>
```

Run the application, and you should see My To Do List as the title of the frame, as in Figure 3:

Figure 3. Injecting the title text into the JFrame



This was your first simple use of dependency injection. You "injected" a plain `String` object into the `title` property of your `MainFrame` class. Under the covers, Spring is automatically creating a `String` with the value `My To Do List` and passing that as a parameter to the `setTitle()` method of the `todo.ui.MainFrame` class. This is an important concept, because all other types of dependency injection in Spring are basically the same. You use the bean definition file(s) to define values, objects, or collections of objects that will be passed (injected) as properties into other objects. Then, Spring wires them together at run time by handling the object creation and property setting for you.

Having the framework do the work of wiring things together is powerful. If you need to change the way that things are wired together in the future, you might only need to change the Spring configuration files and not need to touch your thoroughly tested and bug-free code. Of course, integration problems can always occur when you rewire existing components together in different ways. However, if your components are designed according to good object-oriented principles such as loose coupling and high cohesion (more on this in the next section, Creating the to-do list: Making a reusable component and showing data in a table), you might find that these types of integration problems are relatively rare.

Creating the to-do list: Making a reusable component and showing data in a table

In this section, you'll start fleshing out the application. You won't see any data in your list until the end of the section, but please be patient; you first must create some layout plumbing so you'll have a place to put your data. It will be worth the wait. Also, this section takes some extra time to create a reusable component, show how it can be reused by Spring without changing it, and discuss the benefits of creating reusable components.

Creating a reusable panel

Now, you'll add a panel to the `MainFrame`. I'm going to cheat a little bit here and predict the future. I know you'll be reusing this panel, so you'll make it generic enough to be reused. Doing so will let you see later in this section (in Adding a table, and reusing the panel) how easy it is to reuse existing classes in Spring, as long as they are designed with reuse in mind. Usually you'd code a more customized and simple class initially, and then make it more generic for reuse later. (Even then, you'd do so only when you are *sure* you need to reuse it. See "You Aren't Going To Need It" in Resources.) However, for the purposes of this tutorial, you'll make it reusable from the start keep things simple and save time.

Make a class called `BoxLayoutPanel` in the `todo.ui` package. `BoxLayoutPanel` is a subclass of JPanel and has a BoxLayout (see Listing 13):

Listing 13. src/todo/ui/BoxLayoutPanel.java

```
package todo.ui;

import java.awt.Component;
import java.util.Iterator;
import java.util.List;
import javax.swing.BoxLayout;
import javax.swing.JPanel;


public class BoxLayoutPanel extends JPanel {
    /**
     * We can't use "components" as the property name,
     * because it conflicts with an existing property
     * on the Component superclass.
     */
    private List panelComponents;
    private int axis;

    public void setAxis(int axis) {
        this.axis = axis;
    }

    public void setPanelComponents(List panelComponents) {
        this.panelComponents = panelComponents;
    }

    public void init() {
        setLayout(new BoxLayout(this, axis));

        for (Iterator iter = panelComponents.iterator();
             iter.hasNext();) {
            Component component = (Component) iter.next();
            add(component);
        }
    }
}
```

As you can see in Listing 13, the `axis(X/Y)` property of the `BoxLayout` is settable with a `setAxis()` method. The panel has a List of `Component`s, which will be automatically added to the panel when it is initialized. The code to set up the `BoxLayout` and add the `component`s to the panel are in an `init()` method that Spring calls when the bean is created, just like the `MainFrame` bean.

Wiring beans together

Now, you need to wire the panel into Spring and provide a component to add to it, so you'll modify app-context.xml as in Listing 14 (by adding the boldfaced elements):

Listing 14. src/todo/app-context.xml - Wiring the BoxLayoutPanel into Spring

```
<bean id="mainFrame" class="todo.ui.MainFrame" init-method="init">
  <property name="contentPane">
    <ref bean="mainPanel"/>
  </property>
  <property name="title">
    <value>My To Do List</value>
  </property>
</bean>
<bean id="mainPanel" class=
   "todo.ui.BoxLayoutPanel" init-method="init">
  <property name="axis">
    <!--  "1" corresponds to BoxLayout.Y_AXIS -->
    <!--  Spring can access constants, but it's more complex -->
    <value>1</value>
  </property>
  <property name="panelComponents">
    <list>
      <ref bean="itemScrollPane"/>
    </list>
  </property>
</bean>
<bean id="itemScrollPane" class="javax.swing.JScrollPane">
</bean>
```

There are a few things to notice in Listing 14. First, you create the `mainPanel` bean much like the `mainFrame` bean. Then, you *inject* it into the `mainFrame` bean, by using the `contentPane` property and `<ref bean="mainPanel"/>`. `setContentPane()` is a method to add a panel to a frame. It is available to you automatically, because `MainFrame` subclasses `JFrame` and therefore inherits the `setContentPane()` method. `<ref bean="mainPanel"/>` simply takes the `mainPanel` object that Spring creates and passes it to the `setContentPane()` method on `MainPanel`.

You also inject values into the `axis` and `panelComponents` properties of `BoxLayoutPanel`. For `axis`, the `1` corresponds to the `Y_AXIS` constant of `BoxLayout`. (Note that Spring does provide a way to set a bean property from a static field value, by using a FieldRetrievingFactoryBean. But I don't want to get too fancy, so you'll just hard-code the value of the `axis` constants.)

The `panelComponents` property takes a `List`, so you use the `<list>` element to make Spring automatically create an `ArrayList` for you. Spring can automatically create the following collection types: `List`, `Set`, `Map`, and `Properties`.

Finally, you defined a third bean, `itemScrollPane`, which is a JScrollPane, so that you have a `Component` in your `List`. `itemScrollPane` doesn't require any custom code, so you don't even need to subclass `JScrollPane`; you just let Spring instantiate it directly. See how you can use Spring to wire together instances of preexisting classes that know nothing about Spring, using their existing methods?

Run the application, and you should see ... pretty much the same thing you saw before. Even though you added a `Panel` and a `ScrollPane`, Swing doesn't show scrollbars unless there is something to scroll. You'll add a component to the `ScrollPane` in the next section. Bear with me -- this will be a fully functional application, once you get through some of these inevitable boring bits.

Adding a table and reusing the panel

Next, you'll add a JTable named `itemTable` to the existing `itemScrollPane`. You'll also place another `BoxLayoutPanel`, named `buttonPanel`, in the existing `mainPanel`. This gives you a panel to hold the buttons. This time, you aren't writing any new Java code, but you are just defining and wiring together more beans in app-context.xml, using existing classes (see Listing 15):

Listing 15. src/todo/app-context.xml - Reusing BoxLayoutPanel

```
<bean id="mainPanel" class="todo.ui.BoxLayoutPanel"
      init-method="init">
  <property name="axis">
    <!--  "1" corresponds to BoxLayout.Y_AXIS -->
    <!--  Spring can access constants, but it's more complex -->
    <value>1</value>
  </property>
  <property name="panelComponents">
    <list>
      <ref bean="itemScrollPane"/>
      <ref bean="buttonPanel"/>
    </list>
  </property>
</bean>
<bean id="itemScrollPane" class="javax.swing.JScrollPane">
  <constructor-arg>
    <ref bean="itemTable"/>
  </constructor-arg>
</bean>
<bean id="itemTable" class="javax.swing.JTable">
</bean>
<bean id="buttonPanel" class=
   "todo.ui.BoxLayoutPanel" init-method="init">
  <property name="axis">
    <!--  "0" corresponds to BoxLayout.X_AXIS -->
    <value>0</value>
  </property>
  <property name="panelComponents">
    <list>
    </list>
  </property>
</bean>
```

Remember how you made the `BoxLayoutPanel` generic so you could reuse it (see Creating a reusable panel)? The `buttonPanel` is an instance of `BoxLayoutPanel` just like `mainPanel`. However, it holds buttons instead of a scroll pane, and it will lay out along the X axis rather than the Y axis -- but this is all configured via Spring, so you don't need to change the code. Reuse without needing to change code is good!

The fact that the `javax.swing.BoxLayout` X_AXIS and Y_AXIS constant values are used as parameter values to the `setAxis()` on `todo.ui.BoxLayoutPanel` is a bit ugly, because it would break our `todo.ui.BoxLayoutPanel` in the unlikely event that `javax.swing.BoxLayout` changed the value of its constants. This means that any class that uses the `setAxis()` method on `todo.ui.BoxLayoutPanel` has an implicit dependency on `javax.swing.BoxLayout`, which is otherwise encapsulated in `todo.ui.BoxLayoutPanel`. This results in increased and unnecessary coupling. However, I exposed the constant value only to keep the code concise for this tutorial; it could easily be cleaned up.

Take a minute to think about how you made the `BoxLayoutPanel` reusable. First, you were careful of your dependencies. The only requirements of the `BoxLayoutPanel` interface are the `axis` value and a `List` of `Component`s. (Almost every Swing GUI element inherits from `Component`.) You could have originally created a `ScrollPaneAndButtonPanePanel` and had specific `setScrollPane` and `setButtonPane` setters for each of those components, but that wouldn't have been very reusable at all. Instead, you just took a generic `List` of `Component`s and iterated over it (facilitated by BoxLayout's linear layout scheme). This means that this pane can be reused to lay out any number of GUI components in a vertical or horizontal line, and it never needs to know what type of components they are. This illustrates the good programming practices of *low coupling* and *high cohesion* in action (see Resources). Your `BoxLayoutPanel` lays out components in a row or column, does it very well, and doesn't do anything else (high cohesion). And, it neither knows nor cares about what it lays out, other than the fact that it is a `List` of `Component`s (low coupling).

Another interesting addition to app-context.xml in Listing 15 is the use of the `<constructor-arg>` element. This is *constructor injection*, as opposed to *setter injection*, which you've been using so far. This sounds kind of complex, but it's really not. You are just passing in (*injecting*) your dependency through a constructor argument, rather than a setter (*accessor*) method argument. The only reason you use it here is that `JScrollPane` lets you add a component only through the constructor; you can't add it through a setter.

Run the application, and you should see -- nothing again. Don't despair -- I promise something more exciting will happen very soon.

To summarize, you now have `MainFrame`, which contains a `BoxLayoutPanel`. The `BoxLayout` panel contains a `JScrollPane` with an empty `JTtable`, and another `BoxLayoutPanel` for the soon-to-be-added buttons. However, because these are all container-type components that don't contain anything, they all just show up as a big gray box. But now that all that plumbing is out of the way, you can add some goodies to create something much more exciting.

Defining a table model

In the next section (Creating the to-do list: Finishing up -- buttons and listeners), you'll display some items in the to-do list. To accomplish this, you first need to create an implementation of TableModel, which you'll set on your `JTable`. Your `TableModel` implementation will be called `ItemTableModel`, and it will simply be a wrapper for a `List` of items. To make this easier, you'll extend the AbstractTableModel abstract class provided by Swing and override only the methods you need to in order to make your `TableModel` behave properly when you add it to the `JTable`. Listing 16 shows the code for `ItemTableModel`:

Listing 16. src/todo/ui/ItemTableModel.java

```
package todo.ui;

import java.util.List;
import javax.swing.table.AbstractTableModel;


public class ItemTableModel extends AbstractTableModel {
    List itemList;
    public boolean isCellEditable(int rowIndex, int columnIndex) {
        return true;
    }

    public int getColumnCount() {
        return 1;
    }

    public String getColumnName(int column) {
        return "Items";
    }

    public void setItemList(List itemList) {
        this.itemList = itemList;
    }

    public int getRowCount() {
        return itemList.size();
    }

    public void setValueAt(Object value,
                           int rowIndex, int columnIndex) {
        itemList.set(rowIndex, value);
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        return itemList.get(rowIndex);
    }
}
```

The methods in `ItemTableModel` are fairly straightforward and self-explanatory. Cells are always editable. (`isCellEditable()` always returns true.) The table has only one column. The row count is always equal to the list size. The `setValueAt()` and `getValueAt()` methods simply access the element in the underlying list at the specified row index. The `columnIndex` parameter can be ignored because the table has only one column. Finally, there are the property and setter for the `itemList` itself. That's all there is to the class. Save it and make sure it compiles. No need to run the application yet because you haven't yet wired the `ItemTableModel` class into the rest of the application.

Showing items in the list

You might be wondering where the `List` of items will come from. That is a very important question, and the very important answer is: *You don't care at this point.* Right now, the only dependency that your `TableModel` has on the data is that it implements the `List` interface. This means that it could come from anywhere -- a database, a network request, or a simple hard-coded `ArrayList`. (You'll use a simple `ArrayList`.) It doesn't matter, as long as you put the data into some class that implements the `List` interface and inject it into your `TableModel`. This is good for a few reasons:

For links to interesting practical and philosophical discussions about simplicity and the benefits of delaying the implementation of details, see "You Arent Gonna Need It" and "Do The Simplest Thing That Could Possibly Work" in Resources.

- You can finish coding and testing your list-display GUI functionality against a *simple* hard-coded list, and *put off until later* the details such how you will store the list or retrieve stored lists.

- The "real" implementation of your list could be developed by someone else, in a completely different project, with his or her own Spring configuration file, and you could still wire it together with your GUI later to make a single application. Spring lets you have multiple app-context bean definition files.
- If you do want to add fancy stuff later to your list implementation, such as database persistence (JDBC or ORM) or remote access (Spring supports several different remoting technologies), you can. The Spring framework is filled with lots of helper and wrapper classes for services such as these, as well as examples of how to use them. This can make your development effort much easier (and require much less code) than if you had to use the APIs of the services directly. And the Spring wrappers are designed to be loosely coupled and reusable just like the beans in your to-do list application.
- Even if you eventually do write a real implementation of the list, you can still plug in a stub implementation at any time it is convenient -- for example, during tests, when you don't want to incur the overhead of a database hit, or when you want to have complete and easy control over the list's behavior (see Resources for more on test-driven development).

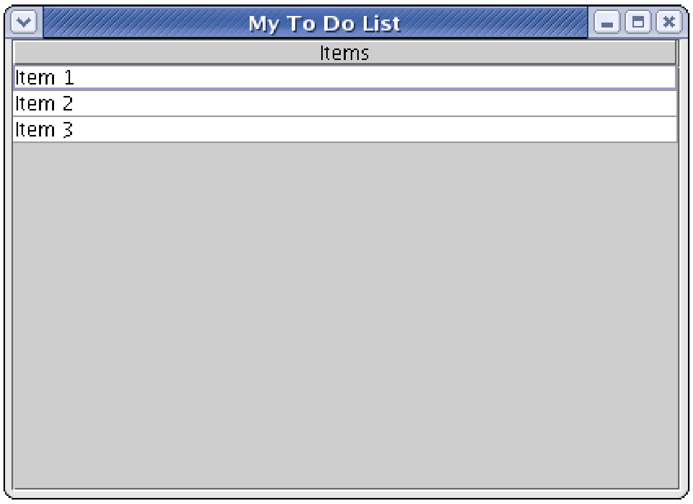
But enough with the gratuitous gushing. Back to the code. It's time to make the stub list implementation and wire it up with your `TableModel` and `JTable`. The define-and-inject pattern for configuring beans in the Spring app-context.xml configuration file should start becoming more familiar by now (see Listing 17):

Listing 17. src/todo/app-context.xml - Adding a stub list implementation

```
<bean id="itemTable" class="javax.swing.JTable">
  <property name="model">
    <ref bean="itemTableModel"/>
  </property>
</bean>
<bean id="itemTableModel" class="todo.ui.ItemTableModel">
  <property name="itemList">
    <ref bean="itemList"/>
  </property>
</bean>
<bean id="itemList" class="java.util.ArrayList">
  <constructor-arg>
    <list>
      <value>Item 1</value>
      <value>Item 2</value>
      <value>Item 3</value>
    </list>
  </constructor-arg>
</bean>
```

Now, run the application, and you should see your data in the list, as shown in Figure 4:

Figure 4. A list with data in it



The list is even editable, because you always return true from `isCellEditable`. Try it: just click in a cell and type or delete characters. When you get bored with that, move on to the next section, where you'll add buttons for adding and deleting items from the list.

Creating the to-do list: Finishing up with buttons and listeners

You'll finish writing your to-do list application in this section. You'll create buttons and listeners that let you add and delete items from the list. The section also includes some discussion about how your button and listener code is different from most other Swing sample code or tutorials.

Creating buttons and listeners

You're in the home stretch now. All you need to do is create Add New and Delete buttons and hook them up to the list using events and listeners. This is all fairly basic stuff in Swing/Java programming (see Resources for good examples and tutorials on these topics), so I'll cover it pretty quickly -- especially because the Spring wiring code is pretty similar to what you've already seen. The classes you'll create to make the buttons work use Swing's implementation of the *Observer pattern*. If you don't understand that pattern you might get a bit confused; see Resources for more information on the Observer pattern.

First, create a subclass of JButton called `ActionListenerButton`, which you'll use for both the Add New and Delete buttons (see Listing 18):

Listing 18. src/todo/ui/button/ActionListenerButton.java

```
package todo.ui.button;

import java.awt.event.ActionListener;
import javax.swing.JButton;


public class ActionListenerButton extends JButton {
    private ActionListener actionListener;

    public void setActionListener(ActionListener actionListener) {
        this.actionListener = actionListener;
    }

    public void init() {
        this.addActionListener(actionListener);
    }
}
```

You might notice one difference between the code you write in this section and many other Swing examples. In your code, the `JButtons`, `ActionListeners`, and the `Components` they interact with are all defined in separate classes and files. In other examples, these might all be coded together in a single file, using anonymous inner classes. The Java API even has an EventHandler class that makes this approach easier. Given that, some people might consider it overkill to create separate classes manually. But others might find it useful, especially if the classes will eventually contain more-complex logic that could benefit from being unit tested in isolation. Also, the more functionality you put into a single class, the more you risk that class having low cohesion. It's a matter of preference. At any rate, splitting the logic into several small classes will help you understand how the different parts work together in this tutorial, especially if you're unfamiliar with the Observer pattern. The important thing is that you can still use Spring to wire your Swing classes together, regardless of the granularity with which you group them into files.

The `ActionListenerButton` class has a property to hold an `ActionListener`, and an `init()` method (called by Spring upon bean creation) that adds the `ActionListener` to the button. This class plays the role of the *subject* in the Observer pattern, and it will notify its `ActionListener` when clicked. Notice that `ActionListenerButton` is in a new `todo.ui.button` package.

Next, create an abstract superclass called `ListTableActionListener`, which contains the common functionality of the ActionListeners for your buttons. This common functionality is the existence of properties for the `list` and `table`, which the subclasses will access and modify when their `actionPerformed()` methods are invoked. The subclasses of `ListTableActionListener` act as the *observers* in the Observer pattern. Listing 19 shows the code for `ListTableActionListener`:

Listing 19. src/todo/ui/button/ListTableActionListener.java

```
package todo.ui.button;

import java.awt.event.ActionListener;
import java.util.List;
import javax.swing.JTable;


public abstract class ListTableActionListener
                      implements ActionListener {
    protected JTable table;
    protected List list;

    public void setList(List list) {
        this.list = list;
    }

    public void setTable(JTable itemTable) {
        this.table = itemTable;
    }
}
```

Next, create `AddNewButtonActionListener`, shown in Listing 20:

Listing 20. src/todo/ui/button/AddNewButtonActionListener

```
package todo.ui.button;

import java.awt.event.ActionEvent;


public class AddNewButtonActionListener extends
  ListTableActionListener {
    public void actionPerformed(ActionEvent e) {
        list.add("New Item");
        table.revalidate();
    }
}
```

`AddNewButtonActionListener` adds a new default item to the list when `actionPerformed()` is invoked and calls `table.revalidate()` to make the table display the newly inserted data.

Finally, create `DeleteButtonActionListener`, shown in Listing 21:

Listing 21. src/todo/ui/button/DeleteButtonActionListener

```
package todo.ui.button;

import java.awt.event.ActionEvent;


public class DeleteButtonActionListener
             extends ListTableActionListener {
    public void actionPerformed(ActionEvent e) {
        int selectedRow = table.getSelectedRow();

        if (selectedRow == -1) {
            // if there is no selected row, don't do anything
            return;
        }

        if (table.isEditing()) {
            // if we are editing the table, don't do anything
            return;
        }

        list.remove(selectedRow);
        table.revalidate();
    }
}
```

This `ActionListener` gets the index of the currently selected row, if there is one, and removes the corresponding item from the list if the row is not currently being edited. After removing the item, the table is revalidated to display the changes.

Wiring in the buttons and listeners

Now that you have the code for the buttons and `ActionListener`s, you wire them together with Spring, just like the other beans, in the app-context.xml file (see Listing 22):

Listing 22. src/todo/app-context.xml - Wiring in the buttons and listeners

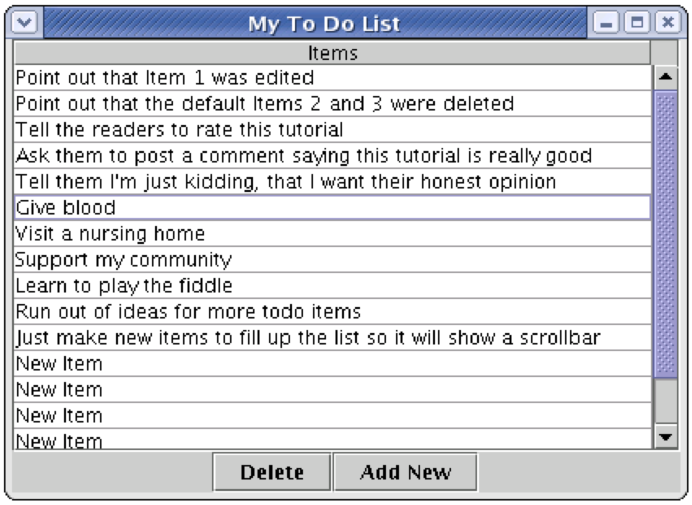
```
    <bean id="buttonPanel" class="todo.ui.BoxLayoutPanel" init-method="init">
    <property name="axis">
      <!--  "0" corresponds to BoxLayout.X_AXIS -->
      <value>0</value>
    </property>
    <property name="panelComponents">
      <list>
        <ref bean="deleteButton"/>
        <ref bean="addNewButton"/>
      </list>
    </property>
  </bean>
  <bean id="deleteButton" class="todo.ui.button.ActionListenerButton"
        init-method="init">
    <property name="actionListener">
      <ref bean="deleteButtonActionListener"/>
    </property>
    <property name="text">
      <value>Delete</value>
    </property>
  </bean>
  <bean id="deleteButtonActionListener"
        class="todo.ui.button.DeleteButtonActionListener">
    <property name="list">
      <ref bean="itemList"/>
    </property>
    <property name="table">
      <ref bean="itemTable"/>
    </property>
  </bean>
  <bean id="addNewButton" class="todo.ui.button.ActionListenerButton"
        init-method="init">
    <property name="actionListener">
      <ref bean="addNewButtonActionListener"/>
    </property>
    <property name="text">
      <value>Add New</value>
    </property>
  </bean>
  <bean id="addNewButtonActionListener"
        class="todo.ui.button.AddNewButtonActionListener">
    <property name="list">
```

```
      <ref bean="itemList"/>
    </property>
    <property name="table">
      <ref bean="itemTable"/>
    </property>
  </bean>
```

You now have two button beans, `addNewButton` and `deleteButton`, both of which are instances of `ActionListenerButton`. Two `ActionListener` beans (`deleteButtonActionListener` and `addNewButtonActionListener`) are defined. They each have the `list` and `table` injected into them and are in turn injected into the buttons. The button beans themselves are added to the `buttonPanel` bean's list of components, so it will lay them out. Note that both the `deleteButtonActionListener` and `addNewButtonActionListener` beans have the same Singleton instances of the `list` and `table` beans injected into them. This means that they will both modify the same `list` and `table` objects.

That's it! Compile and run, and you should see something similar to Figure 5:

Figure 5. The completed To Do List application


My To Do List
Items
Point out that Item 1 was edited
Point out that the default Items 2 and 3 were deleted
Tell the readers to rate this tutorial
Ask them to post a comment saying this tutorial is really good
Tell them I'm just kidding, that I want their honest opinion
Give blood
Visit a nursing home
Support my community
Learn to play the fiddle
Run out of ideas for more todo items
Just make new items to fill up the list so it will show a scrollbar
New Item
New Item
New Item
New Item
Delete
Add New

Try out the **Add New** and **Delete** buttons. Make sure you try adding enough rows and/or resizing the window so that you can see the scrollbar working. You now have a fully functional, rich-client application, written using Spring and Swing. There are a few bugs hiding in it, resulting from this simple implementation. You get extra credit if you find them, and double extra credit if you fix them. If you want to see a much more production-quality example of a Swing application written using Spring, continue on to the next section and check out the Spring Rich Client Project.

The Spring Rich Client Project

This section introduces you to the Spring Rich Client Project (RCP). Although it is still in prerelease development at the time this tutorial is being written, it is a very interesting example of Spring being used to build sophisticated Swing applications.

A slightly higher skill level is required for this section because you'll need to obtain and run (and possibly build) the project yourself. Currently, JDK 1.5 is required to run the sample application.

Also, be aware that latest source might have some temporary problems building or running because it is in still active development.

Spring Rich Client Project overview

The main Spring framework project has a subproject called The Spring Rich Client Project. It is hosted as a SourceForge project and has an active mailing list (see Resources). Although this project is still in a prerelease stage and undergoing architectural changes, it is functional and usable.

The Spring RCP is a good example of the Spring framework's flexibility and of the way Spring provides building blocks that can be used as a foundation for more complex applications and frameworks. It is a basic framework for building Swing-based GUI applications. By providing high-level abstractions such as commands, lifecycle, rules, wizards, forms, views, and preferences, it lets you create nice-looking GUI components and event handling without too much low-level code. A "Petclinic" sample application bundled with the RCP serves as a good example.

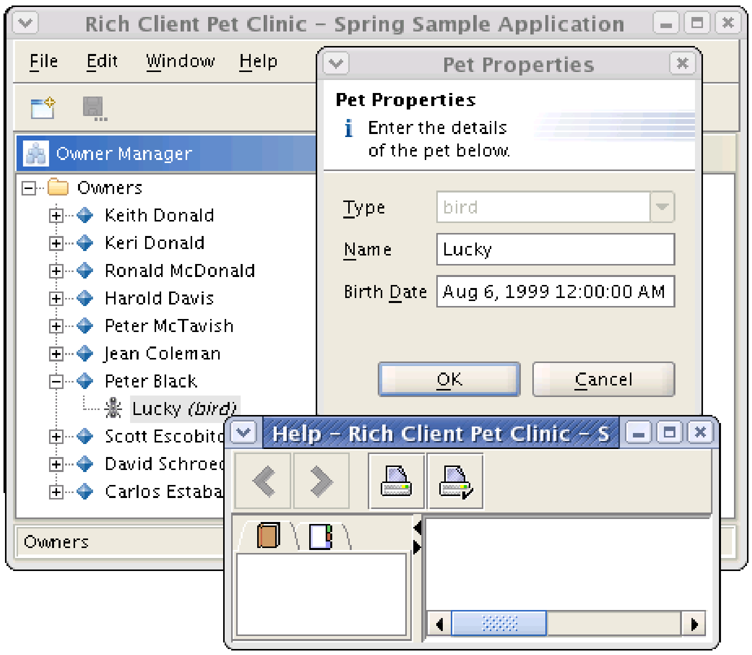
At the time this tutorial is being published, no distributed release package of Spring RCP is available. Recent copies of the RCP binaries and source are available at a hosting site (see Resources). If you can't get it there, or you want the latest source, you need to check it out from the Spring RCP CVS repository and build it.

To get the RCP source from CVS, visit the CVS page for the RCP on SourceForge, and check out the HEAD (see Resources). If you don't know how to use CVS, you can visit the CVS home page and/or download a GUI CVS client such as TortoiseCVS or Cervisia (see Resources). If you use Eclipse, then you can use the CVS Repository Exploring perspective to check out the code.

Once you have the code checked out, find the Ant script at samples/petclinic/build.xml and run the `build-standalone` target. Then run the commands in samples/petclinic/bin/petclinic-standalone.bat. If you have problems and you checked out the source, you might need to rebuild using the `clean` and `alljars` targets in the main build.xml in project's root. If you use Eclipse, run **samples/petclinic/src/ -> org.springframework.richclient.samples.petclinic.PetClinicStandalone**.

When you run the application, you should get a wizard and then a login screen. Enter any ID and password and click on **Cancel** to get into the application, as shown in Figure 6:

Figure 6. The Spring Rich Client Project Petclinic sample application



Play around with the application a bit and then look into the Petclinic sample source code to see how it works. Even if you don't use the RCP directly, the code can provide you with ideas for designing your own rich-client GUIs using Spring. The RCP and the bundled sample applications also provide good examples of how to use the services provided by the core Spring framework, including event handling, remoting with Hessian, JDBC datasource access, and transaction management with AOP. Just as in your to-do list application, the XML context files are a good starting place to learn how a Spring application is put together.

Summary

In this tutorial, you created a Swing GUI application from scratch, using the Spring framework. You learned about the use and benefits of Spring and about dependency injection. And you learned a bit about good programming practices, such as creating reusable components, and the benefits of loose coupling and high cohesion. You also took a look at the Spring Rich Client Project, which is an example of a framework to support the construction of Swing-based GUI applications using Spring. To learn more, try using Spring out on your next or current project. As you can see from this tutorial, it's easy to learn and use the basics.

Also, using Spring is noninvasive, so you could easily use it in a new module of an existing application without affecting the rest of the application. If you can create an `ApplicationContext` and get beans from it, then you can use Spring to wire your objects together. Even if you don't care about using dependency injection for your application right now, you can still use the many useful components that are provided as part of the Spring framework.

You don't even need to use an `ApplicationContext` if you don't want to. Instead, you can use the helper and wrapper classes from the framework directly, instantiating and wiring them together manually (although this would probably be harder than just configuring and using them through an

`ApplicationContext`). The point is that you *could* if you wanted to, because of the flexibility and reusability that are designed into all aspects of Spring.

In my experience, using dependency injection is a very different approach to designing applications. It simplifies many things, and it can cause you to come up with new and different designs that are better than you would have otherwise. It is especially powerful if you use unit testing a lot, because it makes it much easier to write loosely coupled, easily testable units of code. Some of the benefits are subtle and take a while to realize, such as the reduced maintenance required because there is less configuration and wiring code to maintain. Don't take my word for it though -- try it out on a project of your own, and please feel free to let me know how it goes. Thanks for reading this tutorial, and remember that all feedback is welcome!

Downloads

Description	Name	Size
Source code for the sample application	j-springswingcode.zip	9KB

Resources

- Spring framework Web site and About Spring: Visit project headquarters for the Spring framework and read the project's mission statement and feature list.
- "What Is Spring, Part 1" and "What Is Spring, Part 2": Excerpts from the book *Spring: A Developer's Notebook* by Bruce A. Tate and Justin Gehtland (O'Reilly, 2005).
- The *Spring series* (Naveen Balani, developerWorks, 2005): A series of articles and examples on the Spring framework.
- "Introduction to the Spring framework": Rod Johnson's article discusses using Spring to develop J2EE applications.
- Spring Rich Client Project home page and developers' mailing list: Learn more about the Spring RCP.
- Swing API Javadoc: Complete documentation for all Swing components.
- Creating a GUI with JFC/Swing: A series of Sun tutorials on how to create GUIs using Swing.
- Singleton Pattern and Observer Pattern: Learn about these design patterns.
- TestDriven.com: Test-driven development (TDD) is a powerful development approach that fits well with Spring.
- "Inversion of Control Containers and the Dependency Injection pattern" and Inversion of Control: Good descriptions of DI/IOC by Martin Fowler.
- "A beginners guide to Dependency Injection": A high-level overview of DI.
- Coupling And Cohesion: Learn more about the good programming practices of low coupling and high cohesion.
- Well-Formed XML Documents: Learn what's involved in maintaining well-formed XML.
- Aspect Oriented Programming with Spring: Documentation on Spring's support for AOP.
- You Arent Gonna Need It (YAGNI): A development approach in which you do not code any functionality until it is needed.
- Do The Simplest Thing That Could Possibly Work: A development approach that encourages you to start new code by keeping it simple and getting it running quickly.
- "An inside view of Observer": An article describing how the Observer pattern is used in Swing.
- Spring download page: Download the Spring framework.
- Eclipse: Download the Eclipse IDE.
- Apache Ant: Download the Ant build tool.
- Apache Maven: Download the Maven build tool and project-management environment.
- RCP hosting site: Download recent RCP binaries and source.
- CVS page for Spring RCP: Get current RCP source code from the project's CVS repository on SourceForge.
- Concurrent Versions System (CVS): CVS is a popular open-source version-control system.
- TortoiseCVS GUI and Cervisia CVS GUI: GUI CVS clients for Windows and Unix, respectively.
- JUnit: Download the popular unit-testing framework. Spring makes it easy to unit test.
- Jemmy: Jemmy is a tool for doing functional testing of Swing GUI applications.
- jMock: jMock is a library for testing Java code using mock objects. Spring, JUnit, jMock, and test-driven development are a powerful combination for writing robust, bug-free code.

About the author

Chad Woolley

Chad Woolley is a software developer living in Tucson, Arizona, in the Sonoran Desert. He has been developing software professionally since the early 1990s, in many different languages and environments from JCL to Java and Rexx to Ruby. He works at Ionami, and has also worked at Video Monitoring Services, ChoiceHotels, and IBM. Chad enjoys writing quality software in efficient ways, and he is a contributor to the open source community. He likes spending time with his family, playing music and singing, hiking, reading, playing video games with his son, and being active in his church. You can reach Chad at thewoolleyweb.com, or email him at thewoolleyman@gmail.com.