CHAPTER 1

A Gentle Introduction to the Spring Framework

The Spring Framework is an open source application framework written in Java, which supports Java 1.3 and later. It makes building business applications with Java much easier compared with using the classic Java frameworks and application programming interfaces (APIs), such as Java Database Connectivity (JDBC) and JavaServer Pages (JSP). Since its introduction, the Spring Framework has significantly improved the way people design and implement business applications by incorporating best-practice methodologies and simplifying development.

As an introduction to the Spring Framework, this chapter will cover the following topics:

- The process of developing a typical business application and the role the Spring Framework can play
- An overview of the modules that make up the Spring Framework
- An introduction to the sample application that you'll be working with in this book
- An example that demonstrates one of the Spring Framework's core features: managing dependencies
- How the Spring Framework integrates with Java Enterprise Edition (Java EE)
- How to set up the Spring Framework in your applications

Building a Business Application

A modern business application typically consists of the following components:

- *Relational database*: Stores the data related to the problem domain. The database is not necessarily part of the application, but the data-access classes have been written for the specific schema of the database, so that the application is closely coupled with the database schema.
- *Graphical user interface (GUI)*: Lets users interact with the business processes that are implemented by the application. Since the days of the web revolution, many business applications are web-based.
- *Business logic*: Controls and monitors the execution of business processes. The business logic must work with the database and is called by the GUI.

Unfortunately, as tens of thousands of Java developers worldwide can testify, developing business applications in Java can be very hard and frustrating. This is especially, although not exclusively, true at the join points, where the business logic meets the database and the GUI meets the business logic.

Java Platform Hurdles

Java is one of the most powerful and easy-to-use programming languages for developing business applications, so it might seem strange to suggest that developing business applications in Java is difficult. The main hurdles involve its extensive set of libraries and frameworks, each of which adds a wide range of capabilities to Java.

The parts of the Java platform that are crucial for building typical business applications are as follows:

- The JDBC API allows Java applications to connect to a wide range of relational databases.
- The Servlet and JSP specifications are crucial for web-based business applications.
- Desktop applications rely heavily on the Swing or Standard Widget Toolkit (SWT) APIs.

Each of these APIs offers useful capabilities for developing business applications, but most of them are very difficult to use. For example, it’s hard to use the JDBC API correctly for very basic queries on a database (see Chapter 5 for an example). JDBC is an *intrusive* API—it influences the design of an application in such a way that the focus of the design shifts away from its original goals toward trying to use the API in the application. In fact, because the JDBC API is so intrusive, application developers should not spend their time trying to use it correctly. The same can be said for many other APIs in the Java platform. This is where the Spring Framework steps in.

Enter the Spring Framework

A new open source application framework for Java was released on the first day of spring 2003. This release was based on the source code introduced in Rod Johnson’s best-selling book, *Expert One-on-One J2EE Design and Development* (Wrox, 2002).

This 1.0 release offered the building blocks for business application development. Common tasks, such as connecting to and querying a database, managing transactions, and configuring applications, were made more accessible and easier to accomplish. These building blocks used the standard Java APIs behind the scenes and spared the developer from handling their complexity. The 1.1 and 1.2 releases consistently improved existing features and added new features and capabilities. The most recent release (2.0) takes the efficiency of the Spring Framework one step further by offering unparalleled improvements to ease of use and functionality.

The Spring Framework has started a revolution in the world of enterprise Java application development and set in motion a series of events that have forever changed the way applications are developed and deployed. A quick look at the modules that make up the framework should give you an idea of its scope.

Introducing the Spring Framework Modules

The Spring Framework is a collection of subframeworks that solve specific problems and are grouped together in modules. You are free to use any of these frameworks separately. Unless otherwise mentioned, these modules are part of the Spring Framework distribution.

Inversion of Control (IoC) Container: Also called the Core Container, creates and configures application objects and wires them together. This means that resources and collaborating objects are provided to objects, so the objects do not need to look them up. This moves an important responsibility out of your code and makes it easier to write and test code. Chapter 2 introduces the Core Container.

Aspect-Oriented Programming (AOP) framework: Works with *cross-cutting concerns*—one solution to a problem that's used in multiple places. The Spring AOP framework links cross-cutting concerns to the invocation of specific methods on specific objects (not classes) in such a way that your code is unaware of their presence. The Spring Framework uses cross-cutting concerns and AOP to let your application deal with transactions without having a single line of transaction management code in your code base. AOP and cross-cutting concerns are covered in Chapters 3 and 4.

Data Access framework: Hides the complexity of using persistence APIs such as JDBC, Hibernate, and many others. Spring solves problems that have been haunting data-access developers for years: how to get hold of a database connection, how to make sure that the connection is closed, how to deal with exceptions, and how to do transaction management. When using the Spring Framework, all these issues are taken care of by the framework. Chapters 5 and 6 cover data access with the Spring Framework.

Transaction Management framework: Provides a very efficient way to add transaction management to your applications without affecting your code base. Adding transaction management is a matter of configuration, and it makes the lives of application developers much easier. Transaction management is quite a complex subject, and in Chapter 7, you'll see how the Spring Framework simplifies it dramatically.

Resource Abstraction framework: Offers a wonderful feature for conveniently locating files when configuring your applications. Chapter 2 discusses resource abstraction.

Validation framework: Hides the details of validating objects in web applications or rich client applications. It also deals with internationalization (i18n) and localization (l10n). Chapter 8 discusses validation.

Spring Web MVC: Provides a Model-View-Controller (MVC) framework that lets you build powerful web applications with ease. It handles the mapping of requests to controllers and of controllers to views. It has excellent form-handling and form-validation capabilities, and integrates with all popular view technologies, including JSP, Velocity, FreeMarker, XSLT, JasperReports, Excel, and PDFs. Chapters 8 and 9 cover the Spring Web MVC and the view technologies.

Spring Web Flow: Makes implementing web-based wizards and complex workflow processes very easy and straightforward. Spring Web Flow is a conversation-based MVC framework. Your web applications will look much smarter once you learn how to use this framework. Spring Web Flow is distributed separately and can be downloaded via the Spring Framework website. *Expert Spring MVC and Spring Web Flow* (Apress, 2006) covers Spring Web Flow in detail.

Acegi Security System: Adds authentication and authorization to objects in your application using AOP. Acegi can secure any web application, even those that do not use the Spring Framework. It offers a wide range of authentication and authorization options that will fit your most exotic security needs. Adding security checks to your application is straightforward and a matter of configuration; you don't need to write any code, except in some special use cases. Acegi is distributed separately and can be downloaded from `http://acegisecurity.org/downloads.html`.

Remote Access framework: Adds client-server capabilities to applications through configuration. Objects on the server can be exported as remotely available services. On the client, you can call these services transparently, also through configuration. Remotely accessing services over the network thus becomes very easy. Spring's Remote Access framework supports HTTP-based protocols and remote method invocation (RMI), and can access Enterprise JavaBeans as a client. *Pro Spring* (Apress, 2005) covers Spring Remoting in detail.

Spring Web Services: Takes the complexity out of web services and separates the concerns into manageable units. Most web service frameworks generate web service end points and definitions based on Java classes, which get you going really fast, but become very hard to manage as your project evolves. To solve this problem, Spring Web Services takes a layered approach and separates the transport from the actual web service implementation by looking at web services as a messaging mechanism. Handling the XML message, executing business logic, and generating an XML response are all separate concerns that can be conveniently managed. Spring Web Services is distributed separately and can be downloaded via the Spring Framework website (`http://www.springframework.org/download`).

Spring JMX: Exports objects via Java Management Extensions (JMX) through configuration. Spring JMX is closely related to Spring's Remote Access framework. These objects can then be managed via JMX clients to change the value of properties, execute methods, or report statistics. JMX allows you to reconfigure application objects remotely and without needing to restart the application.

Introducing the Sample Application

The sample application that comes with this book is a complex business application that tracks the course of tennis tournaments and matches. The application consists of three modules that perform the following functions:

Manage tennis tournaments and players: The application creates tournaments and players in the database and handles player registration for tournaments. The application will automatically place players in tournament pools based on their Association of Tennis Professionals (ATP) ranking and will draw the matches for each pool. The application will also automatically create a calendar for each court that's available during the course of the tournament and manage the many other variables of a tournament.

Track the course of tennis matches played during tournaments: The application has a user interface that records each point and error during the course of a tennis match. The application knows when a set is over, who won it, and when the match is over. The business logic behind this can be easily ported to mobile devices such as cell phones to conveniently track the course of a match from the audience.

Report on historic data: Reports show the tournament history of individual players, the results of individual matches and pools, the consistency of the tennis game of individual players, a consistency comparison between players, and many other interesting pieces of data related to tennis matches.

One of the core functions of the sample application is to track the course of a tennis match. To better understand the domain of this application, you should have a basic understanding of the game of tennis. Tennis has many rules and statistics, but for the sample application, we'll keep the basic rules for the game as follows:

- A player is either the *server* of a game or the *receiver*. The application will automatically rotate the service when a game ends.
- A service that scores a point without the receiver being able to touch the ball is called an *ace*. The number of aces scored by each player in the course of a match is an important statistic to track.

- A server that makes an error during the service gets another chance. If the server fails at the second attempt, the point goes to the receiver. The number of single and double service errors is another important statistic.
- When the receiver handles the ball and returns the service, a *rally* begins. The point goes to the player who can force the opponent to make an error. Some errors cannot be attributed to any factor other than poor judgment by a player or lack of concentration and are called *unforced errors*. This is another important statistic.

To summarize, the application needs to track the following statistics:

- Who scores each point
- The number of aces per player
- The number of single and double service errors per player
- The number of unforced errors per player

The application will use the scores to calculate when a game is over, when a set is over, and when the match is over. The other statistics are stored for each player per each set.

The sample application is web-based and uses the Spring Framework throughout. Its implementation proves that the Spring Framework reduces the indirect costs of development projects by providing solutions to common problems out of the box. In other words, you don't have to reinvent the wheel. This book will use code from the sample application to illustrate how to use the different parts of the Spring Framework. By studying the implementation, you will be able to familiarize yourself with the most efficient usage of the Spring Framework in typical business applications. The sample application comes with extensive documentation that explains the design choices and the usage of the Spring Framework. You can download the sample application and all the examples used throughout the book from the Source Code/Download section of the Apress website (`http://www.apress.com`).

Now that you've seen the application we are going to build, let's look at an important component of application development—managing dependencies—and how the Spring Framework removes a lot of the complexity.

Managing Dependencies in Applications

To demonstrate how the Spring Framework manages dependencies, let's take a look at a use case from the sample application that needs a data-access object that is configured to connect to the database. We'll see how a plain Java application deals with this situation and contrast this with how Spring does it.

A Use Case That Has Dependencies

One of the requirements of the sample application is to start recording the course of a match during a tournament. Before a tournament starts, all players who have registered are divided into pools, depending on their ranking, age, and gender. For each pool, matches are created in the database. If a pool consists of 32 players, 5 rounds are created: 16 matches in the sixteenth final, 8 matches in the eighth final, 4 matches in the quarter final, 2 matches in the semifinal, and 1 final match. The matches of the sixteenth final are drawn at the start of the tournament.

When any match in the pool is started, the application will check in the database for the following information:

- Whether the match exists
- If the match hasn't finished yet
- If there are any previous matches
- If both previous matches have finished and who the winners are

Some matches are not played because one or both players don't show up, give up before they start, or are injured.

The `TournamentMatchManager` interface has a `startMatch()` method that takes the identifier of the match to start, as shown in Listing 1-1.

Listing 1-1. *The TournamentMatchManager Interface*

```
package com.apress.springbook.chapter01;

public interface TournamentMatchManager {
  Match startMatch(long matchId) throws
    UnknownMatchException,
    MatchIsFinishedException,
    PreviousMatchesNotFinishedException,
    MatchCannotBePlayedException;
}
```

This interface defines the contract of `TournamentMatchManager`. Classes that implement this interface must go through all the steps in the process of starting a tennis match, as shown in Listing 1-2.

Listing 1-2. *The DefaultTournamentMatchManager Class, Which Implements TournamentMatchManager*

```
package com.apress.springbook.chapter01;

public class DefaultTournamentMatchManager implements
        TournamentMatchManager {
  private MatchDao matchDao;

  public void setMatchDao(MatchDao matchDao) {
    this.matchDao = matchDao;
  }

  protected void verifyMatchExists(long matchId) throws
          UnknownMatchException {
    if (!this.matchDao.doesMatchExist(matchId)) {
      throw new UnknownMatchException();
    }
  }

  protected void verifyMatchIsNotFinished(long matchId) throws
          MatchIsFinishedException {
    if (this.matchDao.isMatchFinished(matchId)) {
      throw new MatchIsFinishedException();
    }
  }

  /* other methods omitted for brevity */
  public Match startMatch(long matchId) throws
          UnknownMatchException, MatchIsFinishedException,
          PreviousMatchesNotFinishedException, MatchCannotBePlayedException {
    verifyMatchExists(matchId);
    verifyMatchIsNotFinished(matchId);
    Players players = null;
    if (doesMatchDependOnPreviousMatches(matchId)) {
      players = findWinnersFromPreviousMatchesElseHandle(matchId);
    } else {
      players = findPlayersForMatch(matchId);
    }
    return new Match(players.getPlayer1(), players.getPlayer2());
  }
}
```

Let's walk through what the `startMatch()` method in Listing 1-2 does:

1. The database is checked for a match with the given identifier (`verifyMatchExists()`).
2. The database is queried to verify that the match hasn't been played already (`verifyMatchIsNotFinished()`).
3. The database is queried again to check if the match that is about to start depends on the outcome of two previous matches (`doesMatchDependOnPreviousMatches()`).
   - If the match depends on previous matches, the winners are loaded from the database (`findWinnersFromPreviousMatchesElseHandle()`) if those matches have ended. If one or both previous matches have not been played, the match is not started and is marked in the database as over.
   - If the match is played in the first round of the tournament, the players who are drawn to play this match are loaded from the database (`findPlayersForMatch()`).
4. When two players have been found and no exceptions have occurred, a `Match` object is returned to the caller. The `Match` object is used to track the course of this game, and when the match is over, the statistics are saved to the database.

The `startMatch()` method needs an implementation of the `MatchDao` interface that defines the contract for working with the database. Implementation classes of the `MatchDao` interface are responsible for informing the business logic about the current state of the match information in the database. This information is vital to let the business process work correctly. (We use an interface here to loosely couple the business logic to the data-access code, as explained in later sections.) The `MatchDao` interface is shown in Listing 1-3.

Listing 1-3. *The MatchDao Interface That's Responsible for Querying the Database*

```
package com.apress.springbook.chapter01;

public interface MatchDao {
  boolean doesMatchExist(long matchId);

  boolean isMatchFinished(long matchId);

  boolean isMatchDependantOnPreviousMatches(long matchId);

  boolean arePreviousMatchesFinished(long matchId);

  Player findWinnerFromFirstPreviousMatch(long matchId);
  Player findWinnerFromSecondPreviousMatch(long matchId);

  void cancelMatchWithWinner(long matchId, Player player, String comment);

  void cancelMatchNoWinner(long matchId, String comment);

  Player findFirstPlayerForMatch(long matchId);

  Player findSecondPlayerForMatch(long matchId);
}
```

If you look at the course of a tournament as a workflow, you'll see that there's a start and an end. The methods that return Boolean values in Listing 1-3 provide the business logic with information about the current state of the tournament.

The methods that return `Player` objects use the information in the database to determine who won previous matches. The `cancelMatchWithWinner()` and `cancelMatchNoWinner()` methods update the state of the matches in the database.

Classes that implement the `MatchDao` interface need a connection to the database. For this purpose, a data source is used (the `javax.sql.DataSource` interface) that creates a connection to the database on demand. Data sources are discussed in more detail in Chapter 5; for now, you only need to know that the `javax.sql.DataSource` interface is used to create connections to the database.

Let's round up the dependencies in this use case. `DefaultTournamentMatchManager` objects need a collaborating object that implements the `MatchDao` interface to access the database. For the remainder of this chapter, we'll use the `JdbcMatchDao` class as an implementation class. The `JdbcMatchDao` class has a dependency on the `javax.sql.DataSource` interface, as shown in Listing 1-4. `JdbcMatchDao` objects need a `DataSource` object to get a connection to the database.

Listing 1-4. *JdbcMatchDao, Which Implements the MatchDao Interface and Queries the Database*

```
package com.apress.springbook.chapter01.jdbc;

import javax.sql.DataSource;

import com.apress.springbook.chapter01.MatchDao;

import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcMatchDao implements MatchDao {
  private JdbcTemplate jdbcTemplate;

  public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
  }

  public boolean doesMatchExist(long matchId) {
    return 1 == jdbcTemplate.queryForInt(
      "SELECT COUNT(0) FROM T_MATCHES WHERE MATCH_ID = ?",
      new Object[] { new Long(matchId) }
    );
  }
  /* other methods omitted for brevity */
}
```

The code in Listing 1-4 uses `JdbcTemplate` from the Spring Framework. Chapter 6 covers this class in much more detail. For now, you only need to know that it's a convenient way to query the database. The example shows a `SELECT` statement that counts how many matches are found in the database with a given identifier. If exactly one match is found, the match exists in the database. In this use case, it's also possible that zero matches are found.

The next sections will discuss how the dependencies of this use case can be satisfied in typical Java applications.

Dealing with the Dependencies in Plain Java

If `DefaultTournamentMatchManager` was used in a regular Java application—for example, in a Swing application—the objects would probably be created inside the application, as shown in Listing 1-5.

Listing 1-5. *Creating the DefaultTournamentMatchManager and Dependencies in Java*

```
package com.apress.springbook.chapter01.swing_application;

import org.apache.commons.dbcp.BasicDataSource;

import com.apress.springbook.chapter01.Match;
import com.apress.springbook.chapter01.jdbc.JdbcMatchDao;
import com.apress.springbook.chapter01.TournamentMatchManager;
import com.apress.springbook.chapter01.DefaultTournamentMatchManager;

public class SwingApplication {
  private DefaultTournamentMatchManager tournamentMatchManager;

  public SwingApplication(DefaultTournamentMatchManager tournamentMatchManager) {
    this.tournamentMatchManager = tournamentMatchManager;

    /* other code is omitted for brevity */
  }

  public static void main(String[] args) throws Exception {
    BasicDataSource dataSource = new BasicDataSource();
    /* Setting the properties of the data source. */
    dataSource.setDriverClassName(System.getProperty("jdbc.driverClassName"));
    dataSource.setUrl(System.getProperty("jdbc.url"));
    dataSource.setUsername(System.getProperty("jdbc.username"));
    dataSource.setPassword("pass");

    JdbcMatchDao matchDao = new JdbcMatchDao();
    matchDao.setDataSource(dataSource);

    DefaultTournamentMatchManager tournamentMatchManager =
      new DefaultTournamentMatchManager();
    tournamentMatchManager.setMatchDao(matchDao);

    new SwingApplication(tournamentMatchManager);
  }
}
```

The class shown in Listing 1-5 uses the Swing API to create a GUI. To launch the Swing application, you need to pass the property values for the data source as command-line parameters, as follows:

```
java -classpath %CLASSPATH% ➥
com.apress.springbook.chapter01.swing_application.SwingApplication ➥
-Djdbc.driverClassName=org.hsqldb.jdbcDriver ➥
-Djdbc.url=jdbc:hsqldb:hsql://localhost/ ➥
-Djdbc.username=sa -Djdbc.password=pass
```

The highlighted lines in Listing 1-5 show where collaborating objects are passed to satisfy the dependencies in the application. The objects are created and configured by means of glue code.

This use case is reasonably complex, but small compared with the entire sample application. The glue code that sets up the application can be kept in one place because the client application, the business logic class, and the data-access class are loosely coupled via interfaces. But if we added more glue code here, things would start to get out of hand.

Configuring the application via glue code is not consistent, which is best illustrated by how the properties of the data source are configured. The property values are copied from the system properties. An alternative is to load properties from a file. There's no consistent way to set property values, which means the complexity will grow rapidly without persistent efforts on the part of the developers.

The use of glue code to set up the configuration of an application causes another, subtler problem that becomes apparent when we want to run the Swing application with another implementation of the `TournamentMatchManager`. When we test the Swing application, we don't want to depend on the state and availability of the database, the data-access code, or the full business logic implementation in `DefaultTournamentMatchManager`. Instead, we create a dummy or stub implementation that just returns a `Match` object. This implementation takes five minutes to write and is ideal for testing the user interface components. The stub implementation is shown in Listing 1-6.

Listing 1-6. *A Stub Implementation of the TournamentMatchManager Interface for Testing Purposes*

```
package com.apress.springbook.chapter01.test;

import com.apress.springbook.chapter01.TournamentMatchManager;
import com.apress.springbook.chapter01.Match;
import com.apress.springbook.chapter01.Player;
import com.apress.springbook.chapter01.UnknownMatchException;
import com.apress.springbook.chapter01.MatchIsFinishedException;
import com.apress.springbook.chapter01.PreviousMatchesNotFinishedException;
import com.apress.springbook.chapter01.MatchCannotBePlayedException;

public class StubTournamentMatchManager implements TournamentMachtManager {
  public Match startMatch(long matchId) throws
      UnknownMatchException, MatchIsFinishedException,
      PreviousMatchesNotFinishedException, MatchCannotBePlayedException {
    Player player1 = Player.femalePlayer ();
    player1.setName("Kim Clijsters");

    Player player2 = Player.femalePlayer();
    player2.setName("Justine Henin-Hardenne");

    return new Match(player1, player2);
  }
}
```

When we want to use this stub implementation, we cannot start the client with its own `main()` method. Instead, we need to create a new class to launch the client in test mode, as shown in Listing 1-7. Because `SwingApplication` and `TournamentMatchManager` are loosely coupled, we can start the application with different dependencies, but again the lack of a consistent approach is apparent.

Listing 1-7. *A Separate Class That Launches SwingApplication with StubTournamentMatchManager*

```
package com.apress.springbook.chapter01.test;

import com.apress.springbook.chapter01.swing_application.SwingApplication;

public class LaunchTheSwingApplication {
  public static void main(String[] args) {
    new SwingApplication(new StubTournamentMatchManager());
  }
}
```

Looking Up Dependencies with JNDI

The previous example highlights the lack of consistency in the way the application is configured as the biggest problem. We can try to solve part of this problem by using the Java Naming and Directory Interface (JNDI). JNDI is the standard Java way of looking up objects from an application server. The configuration of the objects happens on the application server and clients can look them up.

When we start our application server, we can look up the data source named `env:jdbc/myDataSource`, as shown in Listing 1-8.

Listing 1-8. *Looking Up a Data Source Using JNDI*

```
package com.apress.springbook.chapter01.swing_application;

import javax.sql.DataSource;

import javax.naming.Context;
import javax.naming.InitialContext;

import java.util.Hashtable;

import com.apress.springbook.chapter01.Match;
import com.apress.springbook.chapter01.jdbc.JdbcMatchDao;
import com.apress.springbook.chapter01.TournamentMatchManager;
import com.apress.springbook.chapter01.DefaultTournamentMatchManager;

public class SwingApplication {
  private TournamentMatchManager tournamentMatchManager;

  public SwingApplication(TournamentMatchManager tournamentMatchManager) {
    this.tournamentMatchManager = tournamentMatchManager;

    /* other code is omitted for brevity */
  }

  public static void main(String[] args) throws Exception {
    Hashtable properties = new Hashtable();
    properties.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
    properties.put(Context.PROVIDER_URL,
        "t3://localhost:7001");
    Context ctx = new InitialContext(properties);
    DataSource dataSource = (DataSource)ctx.lookup("env:jdbc/myDataSource");
    JdbcMatchDao matchDao = new JdbcMatchDao();
    matchDao.setDataSource(dataSource);

    DefaultTournamentMatchManager tournamentMatchManager =
      new DefaultTournamentMatchManager();
    tournamentMatchManager.setMatchDao(matchDao);

    new SwingApplication(tournamentMatchManager);
  }
}
```

The problem of setting the data source property values is solved in Listing 1-8, but it's fair to say some difficulties remain:

- We've coupled our Swing application to an application server to get the data source. This seriously limits the deployment options of the application; it now requires a running application server for the application to start.
- The main consistency problem hasn't been solved yet. There's just as much glue code as in Listing 1-5.
- We still need the `TestingTheSwingApplication` class in Listing 1-6 to launch the Swing application in test mode.

■**Note** If you're not familiar with JNDI and application servers, you only need to remember that clients can ask a special Java server process for objects by name. Don't worry if you can't follow the discussion on JNDI in this section. You only need to realize that using the standard Java way of locating objects doesn't solve the inconsistency issue.

Overall, JNDI hasn't added much value to the application compared to the previous solution. In fact, it's hard to say which of the two approaches is preferable, as it's a choice between two evils.

Using JNDI adds a dependency to an application server to the application and does nothing to reduce the glue code. JNDI adds a bit of consistency because we can now change the connection settings of the data source in the application server without affecting our application. Compare this to Listing 1-5 and the command line for launching the Swing application, where we need to pass in the database connection setting via command-line parameters.

Using the Spring Framework to Provide Dependencies

The complex setup of the two previous code examples and the lack of consistency are caused by the fact that we, as developers, are responsible for obtaining the collaborating objects for our application. We need to write code to create objects and look up a data source, which bothers us, because it adds no value to our application. The business logic and data-access code are finished, and the Swing application is ready to use, but before we can use both together, we must write glue code to tell our application how to assemble itself.

The solution is to move the configuration code out of our application and use the Spring Framework to create and assemble the application components. This will free us from writing glue code and gives us a consistent way of configuring our application. The key is to use dependency injection.

Introducing Dependency Injection

Dependency injection (DI) is the core feature of the Spring Framework Core Container. It provides a mechanism to pass, or *inject*, dependencies to objects. Dependency injection is a method of *inversion of control* (IoC). Figure 1-1 shows how IoC and dependency injection relate to each other.

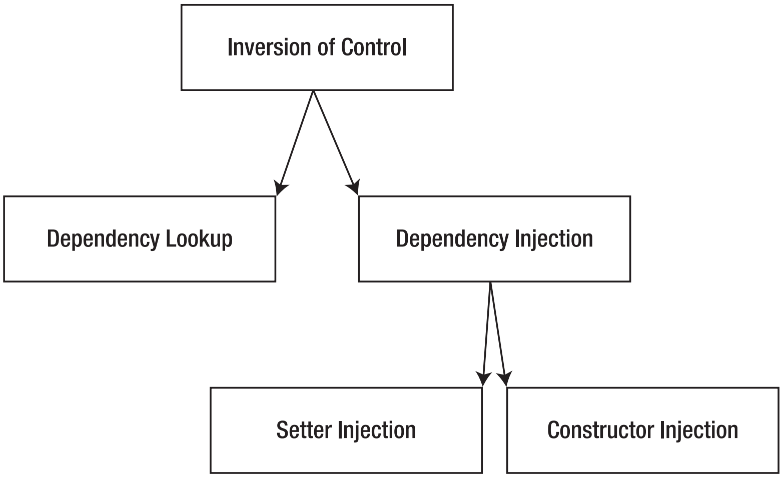

Figure 1-1. *The relationship between IoC and dependency injection*

As shown in Figure 1-1, IoC provides two ways to resolve dependencies: dependency lookup and dependency injection. *Dependency lookup* places the responsibility of resolving dependencies in the hands of the application. The example in Listing 1-8 uses JNDI to obtain a data source, which is a form of dependency lookup. Dependency lookup has been the standard way of resolving dependencies in Java for many years but will always require glue code. The Spring Framework supports dependency lookup, as will be discussed in Chapter 2.

In contrast, dependency injection places the responsibility of resolving dependencies in the hands of an IoC container such as the Spring Framework. A configuration file defines how the dependencies of an application can be resolved. The configuration file is read by the container, which will create the objects that are defined in this file and inject these objects in other application components. The Spring Framework Core Container supports two types of dependency injection to inject collaborating objects: setter injection and container injection.

Setter injection uses `set*()` methods—also called *setter methods*, or *setters* for short—to inject collaboration objects. `set*()` and `get*()` methods together form a JavaBean property, as defined in the JavaBean specifications. To use setter injection, the Spring Framework Core Container must first create an object and then call the setter methods that are defined in the configuration file. Listing 1-9 shows a class with setter methods and a configuration file (you'll see a fuller example of a configuration file in Listing 1-13, later in this chapter). The setter method and the corresponding configuration have been highlighted.

Listing 1-9. *An Example of Setter Injection*

```
package com.apress.springbook.chapter01;

public class DemonstratingBean {
  private String name;
  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return this.name;
  }
}

<!-- Spring Framework configuration file -->
<beans>

  <!--  Injecting Steven in the name property -->
  <bean id="bean" class="com.apress.springbook.chapter01.DemonstratingBean">
    <property name="name" value="Steven"/>
  </bean>
</beans>
```

Constructor injection calls a constructor to inject collaborating objects. The Spring Framework Core Container creates objects and injects collaborating objects at the same time. Listing 1-10 shows a class with a constructor and a configuration file.

Listing 1-10. *An Example of Constructor Injection*

```
package com.apress.springbook.chapter01;

public class DemonstratingBean {
  private String name;
  public DemonstratingBean(String name) {
    this.name = name;
  }

  public String getName() {
    return this.name;
  }
}

<!-- Spring Framework configuration file -->
<beans>

  <!--  Injecting Steven in the constructor -->
  <bean id="bean" class="com.apress.springbook.chapter01.DemonstratingBean">
    <constructor-arg value="Steven"/>
  </bean>
</beans>
```

Both of these techniques lead to the same results from the following piece of code, where we load the configuration file and the injected dependencies with an `XmlBeanFactory` and obtain an instance of the bean:

```
XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("config.xml"));

DemonstratingBean demoBean = (DemonstratingBean)factory.getBean("bean");

System.out.println("Bean property value: " + demoBean.getName());
```

Dependency injection is clearly the preferred way of resolving dependencies, as it enables developers to loosely couple the layers of an application and removes glue code. The result is a cleanly designed application and configuration in one place in a readable and easy-to-change format. In fact, the dependency injection model of the Spring Framework Core Container is a very powerful deployment model. As you create configuration files, you are configuring your application for deployment. Dependency injection is a very important mechanism in the Spring Framework, and the next chapter discusses it in much more detail.

Now, let's continue with the sample use case and see how to handle it with the Spring Framework.

Handling the Use Case with the Spring Framework

The Spring Framework will look at a configuration file (which we need to create) and will automatically create and assemble all objects that are defined in that file. This leaves us with only one task: to bootstrap the Spring Framework and instruct it to read the configuration file and perform the work at hand.

But first, we'll remove all the glue code in our application, as shown in Listing 1-11.

Listing 1-11. *The SwingApplication Class Without Glue Code, Reduced to Its Essence*

```
package com.apress.springbook.chapter01.swing_application;

import com.apress.springbook.chapter01.Match;
import com.apress.springbook.chapter01.TournamentMatchManager;

public class SwingApplication {
  private TournamentMatchManager tournamentMatchManager;

  public SwingApplication(TournamentMatchManager tournamentMatchManager) {
    this.tournamentMatchManager = tournamentMatchManager;

    /* other code is omitted for brevity */
  }
}
```

We've removed the `main()` method as well as the glue code. By looking at the `import` statements of the `SwingApplication` class, we can tell that this class has minimal dependencies on the interfaces of the business logic and the domain classes of the application (`SwingApplication` uses the `Match` class internally). We will come back to this point later in this section, but for now, keep in mind that the `DefaultTournamentMatchManager` and `JdbcMatchDao` classes are not imported anywhere in the application.

Next, we create a skeleton bootstrap class, as shown in Listing 1-12, which will call the API of the Spring Framework. We don't add any implementation to this class for now, but we want to give you a mental hook to where the Spring Framework will fit in the picture.

Listing 1-12. *A Bootstrap Class That Will Launch the Application*

```
package com.apress.springbook.chapter01.spring;

public class SpringBootstrap {
  public static void main(String[] args) throws Exception {

    /* Call the Spring Framework API here! */

  }
}
```

Now we need to create the configuration file that tells Spring which objects to create and how to assemble them. We will use an XML file to hold the configuration instructions. The notation used in this file is defined by the Spring Framework and is consistent for all applications that use it. Listing 1-13 shows the configuration file that will be loaded by the Spring Framework.

Listing 1-13. *The Configuration File That Will Be Loaded by the Spring Framework*

```
<?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="swingApplication"
        class="com.apress.springbook.chapter01.swing_application. ➥
SwingApplication">
    <constructor-arg ref="tournamentMatchManager"/>
  </bean>

  <bean id="tournamentMatchManager"
        class="com.apress.springbook.chapter01.DefaultTournamentMatchManager">
    <property name="matchDao" value="matchDao"/>
  </bean>

  <bean id="matchDao"
        class="com.apress.springbook.chapter01.jdbc.JdbcMatchDao">
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
    <property name="url" value="jdbc:hsqldb:hsql://localhost"/>
    <property name="username" value="sa"/>
    <property name="password" value=""/>
  </bean>
</beans>
```

This configuration file instructs the Spring Framework to create four objects and assemble them. We will discuss this file in detail in the next chapter.

The only thing that remains to be done is to implement the `main()` method of the `SpringBootstrap` class to call the Spring Framework and instruct it to load the configuration file, as shown in Listing 1-14.

Listing 1-14. *The SpringBootstrap Class Uses the Spring Framework to Load a Configuration File*

```
package com.apress.springbook.chapter01.spring;

import java.io.FileInputStream;
import java.util.Properties;

import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class SpringBootstrap {
  public static void main(String[] args) throws Exception {
    /* Check if the location of the configuration file has been passed
    * as an argument.
    */
    if (args.length == 0) {
      throw new IllegalArgumentException("Please provide the location of a " +
        "Spring configuration file as argument!");
      }

      /* Call the Spring Framework API here! */
      XmlBeanFactory factory =
        new XmlBeanFactory(new FileSystemResource(args[0]));

      /* Pause the application until a key is pressed */
      System.out.println("Press any key to close the application");
      System.in.read();

      /* Key has been pressed; close the application and exit */
  }
}
```

We've modified the `SpringBootstrap` class to first check if the location of the configuration file has been passed as a command-line argument. Next, we create the Spring Framework Core Container and tell it to load the configuration file that has been passed as an argument. These are the lines highlighted in Listing 1-14. The next chapter details how to configure the Spring Framework.

When we launch the application, we need to pass the location of the configuration file as a command-line argument, as follows:

```
java -classpath %CLASSPATH% ➥
com.apress.springbook.chapter01.SpringBootStrap ➥
 ./src/java/com/apress/springbook/chapter01/spring/➥
swingApplicationConfiguration.xml
```

Testing the Application

Because we've configured the application components in a Spring configuration file, we can easily use `StubTournamentMatchManager` by creating a new configuration file, as shown in Listing 1-15.

Listing 1-15. *A Configuration File for Testing the Swing Application*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <!-- configuring StubTournamentMatchManager (it doesn't have dependencies) -->
  <bean id="tournamentMatchManager"
        class= "com.apress.springbook.chapter01.test.StubTournamentMatchManager">
  </bean>

  <!-- configuring SwingApplication -->
  <bean id="swingApplication"
        class="com.apress.springbook.chapter01.swing_application.SwingApplication">
    <property name="tournamentMatchManager" ref="tournamentMatchManager"/>
  </bean>
</beans>
```

Next, we can launch the `SpringBootstrap` class with the test configuration file, as follows:

```
java -classpath %CLASSPATH% ➥
com.apress.springbook.chapter01.spring.SpringBootstrap ➥
./src/java/com/apress/springbook/chapter01/spring/test/➥
swingApplicationTestConfiguration.xml
```

We can change the configuration of the application by modifying the configuration file because we use the Spring Framework. The code of our application is not affected.

Reviewing Loosely Coupled Application Layers

This brings the example full circle. As we've stated before, the `DefaultTournamentMatchManager` and `JdbcMatchDao` classes are not imported anywhere in the application. This means the layers of the application are loosely coupled:

- The `SwingApplication` class is part of the presentation layer (it creates the GUI) and has a dependency on the `TournamentMatchManager` interface. This dependency is received through the constructor, as shown in Listing 1-11. The configuration file in Listing 1-13 defines which object will be passed in the constructor.
- The `DefaultTournamentMatchManager` class implements the `TournamentMatchManager` interface and is part of the business logic layer. It has a dependency on the `MatchDao` interface, which is received through the `setMatchDao()` method as shown in Listing 1-2. This type of method is called a setter or setter method, and its purpose is to receive a collaborating object, as demonstrated in Listings 1-5 and 1-8. The configuration file in Listing 1-13 defines which object will be passed to the `setMatchDao()` method.
- The `JdbcMatchDao` class implements the `MatchDao` interface and is part of the data-access layer (it queries the database). It has a dependency on the `javax.sql.DataSource` interface, which is received through the `setDataSource()` method. This is again a setter method that receives a collaborating object. The configuration file in Listing 1-13 defines which object will be passed to the `setDataSource()` method.

One set of classes doesn't fit in any of these layers: the classes of the domain model, such as the `Match` class. These classes do not implement interfaces and are not controlled by the Spring Framework. Domain model classes encapsulate the business rules of the application. If you look back at Listing 1-2, you will notice `DefaultTournamentMatchClass` has a supporting role in the overall application by loading players from the database and creating a `Match` object. Domain model classes are typically used in each layer of the application.

Extending the Application

As other use cases are added to this application, new classes will be added to each layer. However, because we will continue to use clearly defined interfaces comparable to `TournamentMatchManager` and `MatchDao`, the layers of the application will remain loosely coupled.

No implementation classes will be imported anywhere in my application, and the Spring Framework will take care of creating objects and managing the dependencies. As the application grows bigger, the size of the configuration file in Listing 1-13 will also increase. The `SpringBootstrap` class will remain unchanged, no matter how big the application becomes.

However, the clean separation of responsibilities in the application design will remain intact as the configuration grows and the configuration will remain consistent. Compare this with the inconsistent approaches in Listings 1-5 and 1-9. Again, the principle that brings this level of consistency to our application and that's implemented by the Spring Framework is dependency injection.

Integrating the Spring Framework with Java EE

Java EE (formerly J2EE), is an addition to Java Standard Edition that provides APIs that integrate enterprise services in the Java platform. Each enterprise service is a standard defined in specifications that are grouped together under the umbrella of Java EE. Table 1-1 summarizes the enterprise services that are part of Java EE 1.4. Other technologies include accessing mail providers, XML parsing, web services, security, and remote access.

Table 1-1. *Java EE 1.4 Services*

Service	Description
Web application development	The Servlet specifications, under the `javax.servlet.*` package
JavaServer Pages (JSP)	Template technology for rendering X/HTML pages, under the `javax.servlet.jsp.*` and `javax.servlet.jsp.tagext.*` packages
Java Naming and Directory Interface (JNDI)	Directory lookup technology, under the `javax.naming.*` package
Java Transaction API (JTA)	Transaction management abstraction technology supporting distributed transactions, under the `javax.transaction.*` package
Java Messaging Service (JMS)	Message sending and consuming technology, integrating with message queue products, under the `javax.jms.*` package
Enterprise JavaBeans (EJB)	An application model for deploying application components in an environment that transparently configures other parts of the Java EE specifications

The most popular Java EE technologies are servlets, JSP, and EJB. Because Java EE provides a standard way of using enterprise services, developers need to learn only one API.

Spring Framework Integration with Java EE Technologies

The Spring Framework adds one or more extra layers of abstraction on top of the Java EE standards and APIs, either to hide the APIs completely from the developers when it makes sense or to make them easier to use. Some APIs can be completely replaced by alternative solutions, like the EJB specifications, as discussed in the next section.

The following is an overview of how the Spring Framework integrates the most important Java EE APIs to make them less painful to use and more powerful:

JNDI: The Spring Framework has very good JNDI integration in the Core Container. As will be demonstrated in the next chapter, JNDI dependency lookups can be configured in configuration files. The object that's returned by the lookup can then be injected by the Core Container as a collaborating object. Moving dependency lookups out of the application code and into a configuration file helps make existing applications more consistent and easier to maintain.

JTA: Spring's Transaction Management framework fully integrates with the JTA API to support transactions that are orchestrated by an application server. Working with the JTA API is too complicated to allow its usage in application code, so the abstraction offered by the Spring Framework is very useful. Chapter 7 will discuss the JTA API integration in more detail.

JMS: The Spring Framework has excellent integration with the JMS API, which makes it much easier to send and receive messages. Sending messages with JMS requires JNDI lookups that are trivial with the Spring Framework. Sending a message is made very easy by using a helper class that's provided by the Spring Framework. With Spring Framework 2.0, it's now possible to receive messages outside an EJB container with full transactional support. See `http://java.sun.com/products/jms/` for more information about JMS.

Web application development: The Spring Web MVC framework integration with the Servlet API makes it very easy to handle web requests in a consistent way. This framework adds many extensions to the Servlet API that are based on best practices. The Spring Web MVC layer is meant to be very thin and to seamlessly integrate with view technologies. The view abstraction offered by Spring Web MVC and integration with view technologies is unparalleled and uses the Servlet API to hide view rendering details from the developer. Chapter 8 discusses using the Spring Web MVC.

Spring and EJB

EJB integrates with JNDI, JTA, JMS, and other Java EE standards. At the time the Spring Framework was introduced (March 2003), EJB was still the deployment model of choice for many business applications, although its popularity was already declining. At the time of this writing, the EJB specifications are all but dead and buried. The new EJB3 specifications have failed to convince application developers. EJB3 has a very limited dependency injection model and does little to bring consistency to your applications.

EJB and the Spring Framework are often compared because they promise the same thing: *an application deployment model with transparent enterprise service like transaction management, security, messaging, and remote access*.

The Spring Framework matches all the features of EJB3 and goes much further. The Spring Framework is agile and powerful, and reaches out to every kind of application. EJB—including EJB3—caters only to one type of application: centrally deployed business logic running on an application server.

The market has shifted away from intrusive deployment models toward open, flexible, nonintrusive deployment models. That being said, the Spring Framework has excellent integration with EJB3.

The EJB3 specifications consist of two parts: the deployment model and the persistence model. The deployment model manages application components in an *EJB container*. These components are available via JNDI. The Spring Framework has excellent integration with JNDI, so it is easy to acquire references to these objects from within the Spring Framework Core Container and inject these as collaborating objects. Therefore, it is easy to integrate with applications that are deployed in an EJB3 environment. Furthermore, Spring's Transaction Management framework can let objects participate in transactions that are controlled by an application server. This ensures applications deployed with the Spring Framework will work with EJB3 without affecting the business logic code. The Spring Framework also offers integration code for the EJB3 persistence specifications. Chapter 5 discusses this integration in detail.

The Spring Framework also provides excellent integration with older versions of the EJB specifications in the same manner.

Setting Up the Spring Framework in Your Applications

You can download the latest version of the Spring Framework from `http://www.springframework.org`. Make sure you download the distribution archive ending with `with-dependencies`, which contains all the JAR files required by the Spring Framework. This version of the Spring Framework distribution will help you find the JAR files you need to set up your application.

■ **Note** Spring 2.0 was first announced at the first edition of The Spring Experience in Miami, Florida, in December 2005. At that time Rod Johnson, founder of the Spring Framework and CEO of Interface21, announced that Spring 2.0 remains fully backward-compatible with Spring 1.2. Furthermore, Spring 2.0 works with Java 1.3 and beyond and with J2EE 1.2 and beyond. This means you are able to drop the Spring 2.0 JAR in your existing projects, and you won't have a single broken dependency.

The Spring distribution contains the following:

- The Spring JARs for the Core Container, the AOP framework, the Data Access framework, the Transaction Management framework, the Remote Access framework, the JMX framework, and the Spring Web MVC framework
- The complete source code of the Spring Framework
- If you've downloaded the distribution with dependencies, all libraries required by the Spring Framework (available in the `lib` folder)
- Reference documentation in HTML and PDF formats
- Sample applications, including JPetStore, PetClinic, and ImageDb

Of the JAR files, two are most important: `spring.jar` and `spring-mock.jar`. `spring.jar` includes the entire Spring Framework and can be found in the `dist` folder of the Spring distribution, and `spring-mock.jar` includes all classes for writing and running tests and can be found in the `dist/modules` folder.

Add the `spring.jar` archive to the classpath of your application to use the Spring Framework in your application. `spring.jar` has a dependency on the `commons-logging.jar` archive, the Jakarta Common Logging API. This file can be found in the `lib/jakarta-commons` folder of the Spring Framework distribution. If you use Spring 2.0, note that the `spring.jar` archive no longer includes the packages with the integration code for the object-relational mapping (ORM) tools Hibernate 2 and 3, JDO, and Oracle TopLink. These packages are moved to their respective JAR archives in the `dist/extmodules` folder of the Spring 2.0 distribution.

■ **Tip** When setting up the `spring.jar` archive in your application, we encourage you to attach the Spring source in your integrated development environment (IDE). This allows you to easily look inside the classes of the Spring Framework. Our understanding of the Spring Framework has significantly improved by regularly looking inside the framework's source code. Attaching the source code is simply a matter of pointing your IDE to the `src` folder of the Spring Framework distribution.

If you add the `spring-mock.jar` archive to the classpath of your application, make sure you also add the `junit.jar` archive, the JUnit test framework. This file can be found in the `lib/junit` folder of the Spring Framework distribution.

Furthermore, you need to add the JAR files for the other frameworks you want use in your application. The `lib/readme.txt` file that is included in the Spring Framework distribution lists the JAR files you need to include, depending on how you use the Spring Framework in your application.

SPRING SAMPLE APPLICATIONS

The Spring Framework distribution comes with three sample applications:

- *JPetStore*: Based on the original Java Pet Store application, this sample application uses the Spring Framework to demonstrate how a nontrivial web application can be built. For data access, iBatis is used, and two web layers configurations are available: one with Spring Web MVC and one with Struts. This sample also demonstrates the use of Spring Remoting.
- *PetClinic*: This application demonstrates the use of a data-access layer and has implementations using JDBC, Hibernate, Apache OJB, and Oracle TopLink. It uses Spring Web MVC in the web layer and also demonstrates the use of JMX.
- *ImageDb*: This application demonstrates the use of binary large object (BLOB) handling, file upload with Spring Web MVC, and Velocity as a template technology.

The sample applications are useful examples that demonstrate popular usage patterns of various features of the Spring Framework. Each sample application has a `readme.txt` file with a motivation for the sample application and a list of the features demonstrated in the sample.

Summary

In this chapter, you've learned about dependency injection, a straightforward mechanism to acquired dependent objects. This technique will be further examined in the next chapter where the Spring Framework Core Container is introduced.

You've been introduced to the different modules of the Spring Framework and the concepts behind IoC and dependency injection. The remaining chapters of this book discuss many of the modules in further detail. Now is a good time to download the latest distribution of the Spring Framework.