1

Jump Start Spring 2

It is always an exciting time when you first start to use a new software framework. Spring 2, indeed, is an exciting software framework in its own right. However, it is also a fairly large framework. In order to apply it effectively in your daily work you must first get some fundamental understanding of the following issues:

- Why Spring exists
- What problem it is trying to solve
- How it works
- What new techniques or concepts it embraces
- How best to use it

This chapter attempts to get these points covered as quickly as possible and get you using the Spring 2 framework on some code immediately. The following topics are covered in this chapter:

- A brief history of the Spring framework and design rationales
- A typical application of the Spring framework
- Wiring Java components to create applications using Spring
- Understanding Spring's autowiring capabilities
- Understanding the inversion of control and dependency injection
- Understanding the available API modules of Spring 2

Having read this chapter, you will be equipped and ready to dive into specific areas of the Spring framework that later chapters cover.

All About Spring

Spring started its life as a body of sample code that Rod Johnson featured in his 2002 Wrox Press book *Expert One on One Java J2EE Design and Development* (ISBN: 1861007841). The book was published during the height of J2EE popularity. Back in those days, the conventionally accepted way to create a serious enterprise Java application was to use Java 2 Enterprise Edition 1.3/1.4 and create complex software components called Enterprise JavaBeans (EJBs) following the then-current EJB 2.*x* specifications.

> *Appendix B, "Spring and Java EE," provides more insight into how the Spring framework differs from and improves upon Java EE.*

Although popular, creating component-based Java server applications using J2EE was not a fun activity. Constructing EJBs and creating applications out of EJBs are complex processes that involve a lot of tedious coding and require the management of a large body of source code — even for small projects.

Rod's description of a lightweight container that can minimize the complexity of a server-side application construction was a breath of fresh air to the stuffy J2EE development community. Spring — in conjunction with simple yet groundbreaking concepts such as dependency injection (discussed later in this chapter) — captured the imagination of many Java server developers.

Around this body of code was born an active Internet-based development community. The community centered around Rod's company's website (`interface21.com/`), and the associated Spring framework website (`springframework.org/`).

Adoption of the framework continues to rise worldwide as Rod and his group continue to develop the framework and as more and more developers discover this practical and lightweight open-source alternative to J2EE. The software framework itself has also grown considerably, supported by an industry of third-party software add-on components.

Focus on Simplicity

By 2007, version 2 of the Spring framework was released, and the use of the Spring framework in lieu of J2EE for server-side enterprise application development is no longer a notion, but a daily reality practiced throughout the world. Spring's focus on clean separation and decoupling of application components, its lightweight philosophy, and its fanatic attitude toward reducing development complexity have won it a permanent place in the hearts and minds of Java enterprise developers.

Spring has had such a major impact on the developer community that the Java Enterprise Edition expert group actually had to revisit its design. In Java EE 5, the complexity involved in creating EJBs has been greatly reduced in response to infamous user complaints. Many of J2EE's lightweight principles and approaches, including dependency injection, evolved over time.

Applying Spring

As you will discover shortly, creating applications using the Spring Framework is all about gathering reusable software components and then assembling them to form applications. This action of assembling components is called *wiring* in Spring, drawn from the analogy of electronic hardware components. The

wired components can be Java objects that you have written for the application, or one of the many prefabricated components in the Spring API library (or a component from a third-party vendor, such as a transaction manager from Hibernate).

It is of paramount importance, then, to understand how components instantiation and wiring work in Spring. There is no better way to show how Spring object wiring works than through an actual example.

The Spring framework works with modularized applications. The first step is to create such an application. The next section shows you how to start with an all-in-one application and break it down into components. Then you can see how Spring adds value by flexibly wiring together the components.

Creating a Modularized Application

Consider a simple application to add two numbers and print the result.

The entire application can be created within one single class called `Calculate`. The following code shows this monolithic version:

```
package com.wrox.begspring;

public class Calculate {

  public Calculate() {}

  public static void main(String[] args) {
    Calculate calc = new Calculate();
    calc.execute(args);
  }

  private void showResult(String result)  {
    System.out.println(result);
  }

  private long operate(long op1, long op2)  {
    return op1 + op2;
  }

  private String getOpsName() {
    return " plus ";
  }
  public void execute(String [] args)  {
    long op1 = Long.parseLong(args[0]);
    long op2 = Long.parseLong(args[1]);
       showResult("The result of " + op1 +
           getOpsName() + op2 + " is "
           + operate(op1, op2) + "!");

  }

}
```

For example, if you need to perform multiplication instead of addition on the operation, the code to calculate must be changed. If you need to write the result to a file instead of to the screen, the code must be changed again. This application can readily be modularized to decouple the application logic from the

mathematic operation, and from the writer's destination, by means of Java interfaces. Two interfaces are defined. The first, called `Operation`, encapsulates the mathematic operation:

```
package com.wrox.begspring;

public interface Operation {
  long operate(long op1, long op2);
  String getOpsName();
}
```

Decoupling at the Interface

Next, a component that performs addition can be written as the `OpAdd` class:

```
package com.wrox.begspring;

public class OpAdd implements Operation{
  public OpAdd() {}
  public String getOpsName() {
    return " plus ";
  }

  public long operate(long op1, long op2) {
    return op1 + op2;
  }

}
```

And another component that performs multiplication can be written as `OpMultiply`:

```
package com.wrox.begspring;

public class OpMultiply implements Operation {
  public OpMultiply() {}
  public String getOpsName() {
    return " times ";
  }

  public long operate(long op1, long op2) {
    return op1 * op2;
  }
}
```

Note that this refactoring creates two components that can be reused in other applications.

Creating the Result Writer Components

In a similar way, the writing of the result either to the screen or to a file can be decoupled via a `ResultWriter` interface:

```
package com.wrox.begspring;

public interface ResultWriter {
```

```
    void showResult(String result) ;
}
```

One implementation of `ResultWriter`, called `ScreenWriter`, writes to the console screen:

```
package com.wrox.begspring;

public class ScreenWriter implements ResultWriter{
  public ScreenWriter() {}
  public void showResult(String result) {
    System.out.println(result);
  }

}
```

Another implementation of `ResultWriter`, called `DataFileWriter`, writes the result to a file:

```
package com.wrox.begspring;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.PrintWriter;

public class DataFileWriter implements ResultWriter {
  public DataFileWriter() {}

  public void showResult(String result) {
    File file = new File("output.txt");
    try {
    PrintWriter fwriter = new PrintWriter(
        new BufferedWriter(new FileWriter(file)));
    fwriter.println(result);
    fwriter.close();

    } catch (Exception ex) {
      ex.printStackTrace();
    }

  }

}
```

Putting the Application Together

With the `Operation` and `ResultWriter` implementations factored out as reusable components, it is possible to glue a selection of the components together to create an application that adds two numbers and prints the result to the screen. This is done in the `CalculateScreen` class:

```
package com.wrox.begspring;

public class CalculateScreen {
```

```
    private Operation ops = new OpAdd();
    private ResultWriter wtr = new ScreenWriter();

    public static void main(String[] args) {
      CalculateScreen calc = new CalculateScreen();
      calc.execute(args);
    }



    public void execute(String [] args)  {
      long op1 = Long.parseLong(args[0]);
      long op2 = Long.parseLong(args[1]);
         wtr.showResult("The result of " + op1 +
             ops.getOpsName() + op2 + " is "
             + ops.operate(op1, op2) + "!");

    }

  }
```

Mixing and Matching Components

If you need an application that multiplies two numbers and prints the result to a file, you can glue the components together in the manner shown in the `CalculateMultFile` class:

```
package com.wrox.begspring;

public class CalculateMultFile {
  private Operation ops = new OpMultiply();
  private ResultWriter wtr = new DataFileWriter();

  public static void main(String[] args) {
    CalculateMultFile calc = new CalculateMultFile();
    calc.execute(args);
  }

  public void execute(String [] args)  {
    long op1 = Long.parseLong(args[0]);
    long op2 = Long.parseLong(args[1]);
       wtr.showResult("The result of " + op1 +
           ops.getOpsName() + op2 + " is "
           + ops.operate(op1, op2) + "!");

  }

}
```

Thus far, you have seen how to take a monolithic application and refactor it into components. You now have a set of two interchangeable math components: `OpAdd` and `OpMultiply`. There is also a set of two interchangeable result writer components: `ScreenWriter` and `DataFileWriter`.

You will see how the Spring framework can add construction flexibility very shortly. For now, take some time to try out the following base application before proceeding further.

Try It Out Creating a Modularized Application

You can obtain the source code for this example from the Wrox download website (`wrox.com`). You can find the directory under `src/chapter1/monolithic`.

The following steps enable you to compile and run first the monolithic version of the Calculate application, and then a version that is fully modularized.

1. You can compile the all-in-one `Calculate` class by going into the `src/chapter1/monolithic` directory:

```
cd src/chapter1/monolithic
mvn compile
```

2. To run the all-in-one application, use the following command:

```
mvn exec:java -Dexec.mainClass=com.wrox.begspring.Calculate -Dexec.args="3000 3"
```

   This runs the `com.wrox.beginspring.Calculate` class and supplies the two numeric arguments as `3000` and `3`. Among the logging output from Maven, you should find the output from the application:

```
The result of 3000 plus 3 is 3003!
```

3. The modularized version of the application is located in the `src/chapter1/modularized` directory of the source code distribution. Compile the source using the following commands:

```
cd src/chapter1/modularized
mvn compile
```

4. Then run the modularized version of Calculate with the `CalculateScreen` command:

```
mvn exec:java -Dexec.mainClass=com.wrox.begspring.CalculateScreen -Dexec.args="3000 3"
```

You should observe the same output from this modularized application as from the all-in-one application.

Note that you should be connected to the Internet when working with Maven 2. This is because Maven 2 automatically downloads dependencies from global repositories of open-source libraries over the Internet. This can, for example, eliminate the need for you to download the Spring framework binaries yourself.

How It Works

The `mvn` command runs Maven 2. Maven 2 is a project management tool that you will use throughout this book and it is great for handling overall project management. (Appendix A provides more information about Maven 2 and can help you to become familiar with it.)

The refactored version of the Calculate application (`CalculateScreen`) is fully modularized and is easier to maintain and modify than the all-in-one version. Figure 1-1 contrasts the monolithic version of `Calculate` with the modularized version, called `CalculateScreen`.

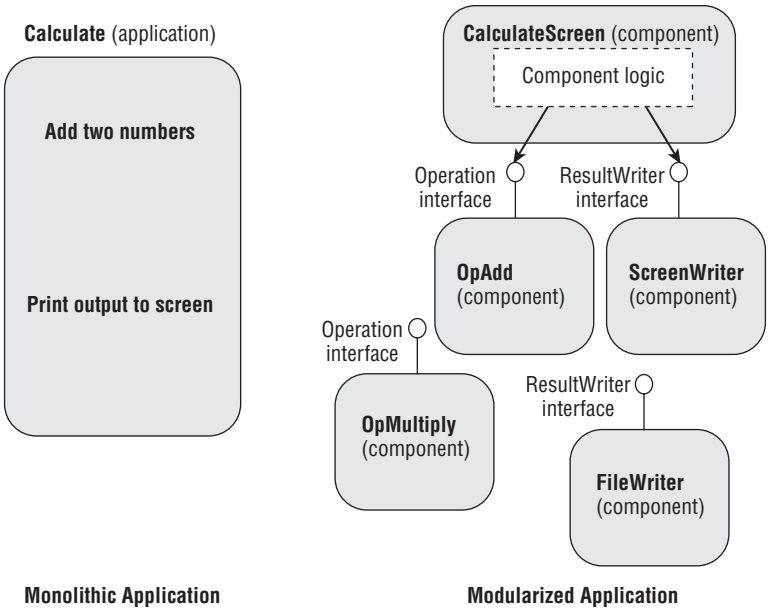

Figure 1-1

In Figure 1-1, there is still one problem: the code of `CalculateScreen` must be modified and recompiled if you need to change the math operation performed or where to display the result. The code that creates instances of `OpAdd` and `ScreenWriter` is hard-coded in `CalculateScreen`. Spring can help in this case, as the next section demonstrates.

Using Spring to Configure a Modularized Application

Figure 1-2 shows graphically how Spring can assist in flexibly interchanging the implementation of the math operation (say, from `OpAdd` to `OpMultiply`) and/or the implementation of `ResultWriter`.

The circle in Figure 1-2 is the Spring container. It reads a configuration file, a context descriptor named `beans.xml` in this case, and then uses the contained information to wire the components together. The context descriptor is a kind of a configuration file for creating applications out of components. You will see many examples of context descriptors throughout this book.

In Figure 1-2, the `CalculateSpring` main class does not directly instantiate the operation or `ResultWriter`. Instead, it defers to the Spring container to perform this task, the instantiation. The Spring container reads the `beans.xml` context descriptor, instantiates the beans, and then wires them up according to the configuration information contained in `beans.xml`.

You can see the code for all of this in the next section.

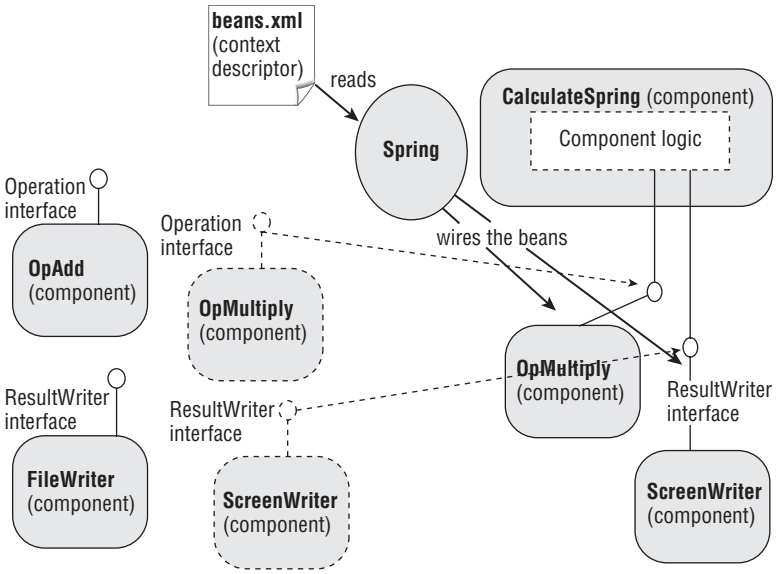

Figure 1-2

Try It Out Compiling Your First Spring Application

The source code for this Spring-wired application can be found in the `src\chapter1\springfirst` directory of the code distribution. Follow these steps to compile and run the application using the Spring container:

Downloading and Installing the Spring Framework

Since Maven 2 downloads dependencies automatically over the Internet, you do not have to download the Spring framework binaries yourself. If you examine the `pom.xml` file in the `src\chapter1\springfirst` directory, you can see that Spring is already specified as a dependency there. (See Appendix A for more information on Maven 2 and `pom.xml`.)

To get all the library modules, dependencies, documentations, and sample code, you can always find the latest version of Spring 2 at `http://www.springframework.org/download`. The code in this book has been tested against the 2.0.6 version of the Spring distribution and should work with all later versions. When selecting the download files, make sure you pick the `spring-framework-2.x.x-with-dependencies.zip` file. This is a significantly larger download, but contains the open-source dependencies that you need. Downloading this file can save you a lot of time downloading dependencies from other locations. To install the framework you need only expand the ZIP file in a directory of your choice. When creating applications using the framework (and not using Maven to manage your builds), you may need to include some of the JAR library files in the bundle (for example, a WAR file for a web application) or refer to them in your build classpath.

1. Change the directory to the source directory, and then compile the code using Maven 2:

```
cd src\chapter1\springfirst
mvn compile
```

   This may take a little while, since Maven 2 will download all the Spring libraries that you need from the global repository. Once the libraries are downloaded, Maven 2 keeps them in your local repository on your computer's hard disk, and you will not have to wait for them again.

2. To run the Springwired version of the modularized application, the `CalculateSpring` class, use the following Maven 2 command:

```
mvn exec:java -Dexec.mainClass=com.wrox.begspring.CalculateSpring -Dexec.args="3000 3"
```

3. Your output from this Spring-wired application should be:

```
The result of 3000 times 3 is 9000!
```

How It Works

The ability to easily wire and rewire reusable Java beans for an application is central to the flexibility offered by the Spring framework.

The `CalculateSpring` main class, instead of instantiating concrete instances of `Operation` or `ResultWriter`, delegates this task to the Spring container. The Spring container in turn reads your configuration file, called the bean descriptor file (or the context descriptor).

In the `CalculateSpring` class, shown here, the highlighted code hooks into the Spring container and tells it to perform the task of wiring together the beans:

```
package com.wrox.begspring;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class CalculateSpring {
  private Operation ops;
  private ResultWriter wtr;

  public void setOps(Operation ops) {
   this.ops = ops;
  }
  public  void setWriter(ResultWriter writer) {
   this.wtr = writer;
  }

  public static void main(String[] args) {
    ApplicationContext context =
      new ClassPathXmlApplicationContext(
            "beans.xml");
```

```
      BeanFactory factory = (BeanFactory) context;
      CalculateSpring calc =
        (CalculateSpring) factory.getBean("opsbean");

      calc.execute(args);
    }



    public void execute(String [] args)  {
      long op1 = Long.parseLong(args[0]);
      long op2 = Long.parseLong(args[1]);
         wtr.showResult("The result of " + op1 +
             ops.getOpsName() + op2 + " is "
             + ops.operate(op1, op2) + "!");

    }

  }
```

The preceding highlighted code creates an `ApplicationContext`. This context is created and provided by the Spring container. In this case, the actual implementation of `ApplicationContext` is called `ClassPathXmlApplicationContext`. The `beans.xml` descriptor is supplied as a constructor argument. Spring's `ClassPathXmlApplicationContext` looks for the instructions for wiring the beans together in the `beans.xml` file, which can be found in the classpath.

ApplicationContext is a BeanFactory

An `ApplicationContext` in Spring is a type of `BeanFactory`. A `BeanFactory` enables you to access JavaBeans (classes) that are instantiated, wired, and managed by the Spring container.

Although there are other `BeanFactory` library classes in Spring, the `ApplicationContext` class is the most frequently used one because it provides a lot of valuable extra features — including support for internationalization, resource loading, integration with external context hierarchies, events publishing, and much more.

> **The `BeanFactory` classes are examples of the *factory method* design pattern. This design pattern enables a framework to provide a means for creating objects without knowing ahead of time the type of object that will be created. For example, the `BeanFactory` in the preceding example is used to create an instance of `CalculateSpring`, a class that `BeanFactory` has no knowledge of.**

Providing the Spring Container with Wiring Instructions

The `ClassPathXmlApplicationContext` constructor takes as an argument the context descriptor file or the bean's wiring file. This file is named `beans.xml` in the example case presented here, but you can use any name you want as long as it has the `.xml` extension, as it is an XML file. This `beans.xml` file is the configuration file describing how to wire together objects. The `beans.xml file` is shown here.

Note the XML schema and namespaces used in the `<beans>` document element. These are standard for Spring 2.0 and the schema defines the tags allowed within the descriptor. You are likely to find these schema in every Spring context descriptor file, except for some pre-2.0 legacy DTD-based descriptor files. (See the sidebar Support of Legacy DTD-Based Spring Wiring Syntax.)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <bean id="screen" class="com.wrox.begspring.ScreenWriter" />
  <bean id="multiply" class="com.wrox.begspring.OpMultiply" />
  <bean id="add" class="com.wrox.begspring.OpAdd" />

  <bean id="opsbean" class="com.wrox.begspring.CalculateSpring">
    <property name="ops" ref="multiply" />
    <property name="writer" ref="screen"/>
  </bean>


</beans>
```

`ClassPathXmlApplicationContext` is part of the Spring container, and it looks for the context descriptor (`beans.xml`) in the Java VM's `CLASSPATH` and creates an instance of an `ApplicationContext` from it. During the instantiation of the `ApplicationContext`, the beans are wired by the Spring container according to the directions within the context descriptor.

Support of Legacy DTD-Based Spring Wiring Syntax

Note that you may also frequently see Spring configuration files using the following document based on document type definition (DTD):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
     "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
...
</beans>
```

This convention is used extensively in versions of Spring before 2.0. XML schema offers far more extensive functionality than DTDs. Although Spring 2 supports DTD-based configurations, it is highly recommended that you use XML schema-based configurations instead. An example of such a configuration can be seen above in the `beans.xml` file.

Creating and Wiring Java Beans

The `<bean>` tag, as its name suggests, is used to instantiate an instance of a bean. The container performs the following actions according to the instructions:

1. Creates an instance of `ScreenWriter` and names the bean `screen`

2. Creates an instance of `OpMultiply` and names the bean `multiply`
3. Creates an instance of `OpAdd` and names the bean `add`
4. Creates an instance of `CalculateSpring` and names the bean `opsbean`
5. Sets the reference of the `ops` property of the `opsbean` bean to the bean named `multiply`
6. Sets the reference of the `writer` property of the `opsbean` bean to the bean named `screen`

Each of these instructions is labeled in bold in the following reproduction of the `beans.xml` context descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  (1)<bean id="screen" class="com.wrox.begspring.ScreenWriter" />
  (2)<bean id="multiply" class="com.wrox.begspring.OpMultiply" />
  (3)<bean id="add" class="com.wrox.begspring.OpAdd" />

  (4)<bean id="opsbean" class="com.wrox.begspring.CalculateSpring">
    (5)<property name="ops" ref="multiply" />
    (6)<property name="writer" ref="screen"/>
  </bean>


</beans>
```

It is very important for you to understand how Java classes are created and wired using a Spring context descriptor. Take a careful look and make sure you see how these actions are carried out.

The net effect is that the `CalculateSpring` logic will be wired with the `OpMultiply` operation and the `ScreenWriter` writer. This is why you see the result of the multiplication on the screen when you run `CalculateSpring`.

Adding a Logging Configuration File

In addition to the `beans.xml`, you also need to create a `log4j.properties` file. The Spring framework uses Apache Commons Logging (about which information is available at `http://jakarta.apache.org/commons/logging/`) to log container and application information. Commons logging can work with a number of loggers, and is configured to work with Apache's Log4j library (an open-source library; information is available at `http://logging.apache.org/log4j/docs/index.html`).

Log4j can read its configuration information from a properties file. In the properties file, you can configure appenders that control where the logging information is written to; for example, to a log file versus to the screen). You can also control the level of logging; for example, a log level of `INFO` prints out a lot more information than a log level of `FATAL`, which only prints out fatal error messages. The following `log4j.properties` file is used by the example(s) and only displays fatal messages to the console. (You can find it in `src\springfirst\src\main\resources`.) Maven automatically copies this file into the correct location and constructs the classpath for the application (via the information provided in the `pom.xml` file). Then Log4J uses the classpath to locate the configuration file.

```
log4j.rootLogger=FATAL, first
log4j.appender.first=org.apache.log4j.ConsoleAppender

log4j.appender.first.layout=org.apache.log4j.PatternLayout
log4j.appender.first.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```

Wiring Beans Automatically by Type

In the preceding example you wired the properties of the `CalculateSpring` bean explicitly. In practice, you can actually ask the Spring container to automatically wire up the properties. The next "Try It Out" shows how to perform automatic wiring.

Try It Out Autowire by Type

Automatic wiring can save you some work when you're creating context descriptors. Basically, when you tell the Spring container to autowire, you're asking it to find the beans that fit together. This can be done automatically by the container without you providing explicit instructions on how to wire the beans together. There are many different ways to autowire, including by name, by type, using the constructor, or using Spring's autodetection. Each of these will be described in a bit. For now, let's discuss the most popular type of autowiring, autowiring by type.

Autowiring by type means that the container should try to wire beans together by matching the required Java class and/or Java interface. For example, a `CalculateSpring` object can be wired with an instance of `Operation` (Java interface) type, and an instance of `ResultWriter` (Java interface) type. When told to autowire by type, the Spring container searches the context descriptor for a component that implements the `Operation` interface, and for a component that implements the `ResultWriter` interface.

This feature can be a time-saver if you are creating a large number of beans in a context descriptor.

To try out autowiring by type with the `CalculateSpring` project, follow these steps:

1. Change the directory to the `src/springfirst/target/classes` directory:

```
cd src/chapter1/springfirst/target/classes
```

2. You should see `beans.xml` here. Maven 2 has copied the context descriptor here during compilation. Modify the `beans.xml` context descriptor, as shown in the following listing; the changed lines are highlighted. Note that you need to remove the lines not shown in the listing.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

<bean id="screen" class="com.wrox.begspring.ScreenWriter" />
<bean id="add" class="com.wrox.begspring.OpAdd" />

<bean id="opsbean" class="com.wrox.begspring.CalculateSpring" autowire="byType" />

</beans>
```

3. Change the directory back to the `/springfirst` directory in which `pom.xml` is located.
4. Now, run the application using Maven 2 with the following command line:

```
mvn exec:java -Dexec.mainClass=com.wrox.begspring.CalculateSpring -Dexec.args="3000 3"
```

First, note that you do not need to recompile at all; you perform this reconfiguration purely by editing an XML file — the context descriptor. Also notice that the output indicates that the `OpAdd` operation and `ScreenWriter` have been wired to the `CalculateSpring` bean. All this has been done automatically by the Spring container:

```
The result of 3000 plus 3 is 3003!
```

How It Works

Even though you have not explicitly wired the `OpAdd` and `ScreenWriter` components to the `CalculateSpring` bean, the container is smart enough to deduce how the three beans must fit together.

This magic is carried out by the Spring container's ability to automatically wire together components. Notice the attribute `autowire="byType"` on the `<bean>` element for the `opsbean`. This tells the Spring container to automatically wire the bean. The container examines the beans for properties that can be set, and tries to match the type of the property with the beans available. In this case, the `CalculateSpring` bean has two property setters:

```
public void setOps(Operation ops) {
   this.ops = ops;
  }
  public  void setWriter(ResultWriter writer) {
   wtr = writer;
  }
```

The first one takes a type of `Operation`, and the second takes a type of `ResultWriter`. In the `beans.xml` file, the only bean that implements the `Operation` interface is the `add` bean, and the only bean that implements the `ResultWriter` interface is the `screen` bean. Therefore, the Spring container wires the `add` bean to the `ops` property, and the `screen` bean to the `writer` property, automatically.

Autowiring can sometimes simplify the clutter typically found in a large descriptor file. However, explicit wiring should always be used if there is any chance of autowiring ambiguity.

You can also autowire by other criteria. The following table describes the other varieties of autowiring supported by Spring 2.

Value of autowire Attribute	Description
`byName`	The container attempts to find beans with the same name as the property being wired. For example, if the property to be set is called `operation`, the container will look for a bean with `id="operation"`. If such a bean is not found, an error is raised.

Continued

Value of autowire Attribute	Description
`byType`	The container examines the argument type of the setter methods on the bean, and tries to locate a bean with the same type. An error is raised when more than one bean with the same type exists, or when there is no bean of the required type.
`constructor`	The container checks for a constructor with a typed argument, and tries to find a bean with the same type as the argument to the constructor and to wire that bean during a call to the constructor. An error is raised when more than one bean with the required type exists, or when there is no bean of the required type.
`autodetect`	The container performs either the constructor or `byType` setter autowiring by examining the bean to be wired. It raises an error if the bean cannot be autowired.
`no`	This is the default behavior when the `autowire` attribute is not specified. The container does not attempt to wire the properties automatically; they must be explicitly wired. This is desirable in most cases if you want explicit documentation of the component wiring.

Understanding Spring's Inversion of Control (IoC) Container

The Spring container is often referred to as an *inversion of control* (IoC) container. In standard component containers, the components themselves ask the server for specific resources. Consider the following segment of code, which can frequently be found in components created for J2EE components:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/WroxJDBCDS");
Connection conn = ds.getConnection("user", "pass");
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery( "SELECT * FROM Cust" );
while( rs.next() )
   System.out.println( rs.getString(1)) ;
rs.close() ;
stmt.close() ;
conn.close() ;
con.close();
```

This code obtains a JDBC (Java Database Connectivity) DataSource from the container using JNDI (Java Naming and Directory Interface) lookup. It is the J2EE-sanctioned way of getting a JDBC connection, and occurs very frequently in component coding. Control for the code is always with the component; in particular the component obtains the DataSource via the following steps:

1. Obtains a new JNDI `InitialContext()` from the container

2. Uses it to look up a resource bound at `jdbc/WroxJDBCDS`
3. Casts the returned resource to a DataSource

When writing your software components to be wired by Spring, however, you do not need to perform the lookup; you just start to use the DataSource. Consider this segment of code from a Spring component:

```
private DataSource ds;
public void setDs(DataSource datasource) {
ds = datasource;
}
...
Connection conn = ds.getConnection("user", "pass");
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery( "SELECT * FROM Cust" );
while( rs.next() )
   System.out.println( rs.getString(1)) ;
rs.close() ;
stmt.close() ;
conn.close() ;
```

This code does exactly the same work as the preceding code segment. However, note that at no time does the component actually ask the container for the DataSource. Instead, the code simply uses the private `ds` variable without knowing how it is obtained.

In this case, control over deciding which DataSource to use is *not* with the component — it is with the container. Once the container has an instance of a DataSource that the component can use, it is placed into the object by calling the `setDs()` method. This is the inversion in IoC! The control for the resource to use is inverted, from the component to the container and its configuration. In this case, the Spring container makes the decision instead of the component.

Of course, in Spring, the DataSource can be wired during deployment via editing of the `beans.xml` descriptor file:

```
<bean name="wroxBean"
      class="com.wrox.beginspring.DataBean">
    <property name="ds" ref="jdbcds"  />
</bean>
```

The preceding code assumes that the `jdbcds` bean is a DataSource that has been wired earlier within the context descriptor. This DataSource instance can be created directly as a `<bean>`, or it can still use JNDI lookup if the container chooses. Spring provides a library factory bean called `org.springframework.jndi.support.SimpleJndiBeanFactory`, if you want to use JNDI lookup.

The key thing to understand is that instead of you having to decide on the mechanism to obtain a resource (such as a JDBC DataSource) in the component code — making it specific to the means of lookup and hard-coding the resource name — using IoC allows the component to be coded without any worry about this detail. The decision is deferred to the container to be made at deployment time instead of at compile time. Figure 1-3 illustrates IoC.

The component on the left side of Figure 1-3 is a standard non-IoC component, and it asks the container for the two resources (beans) that it needs. On the right side of Figure 1-3, the container creates or locates

the required resource and then injects it into the component, effectively inverting the control over the selection of the resource from the component to the container.

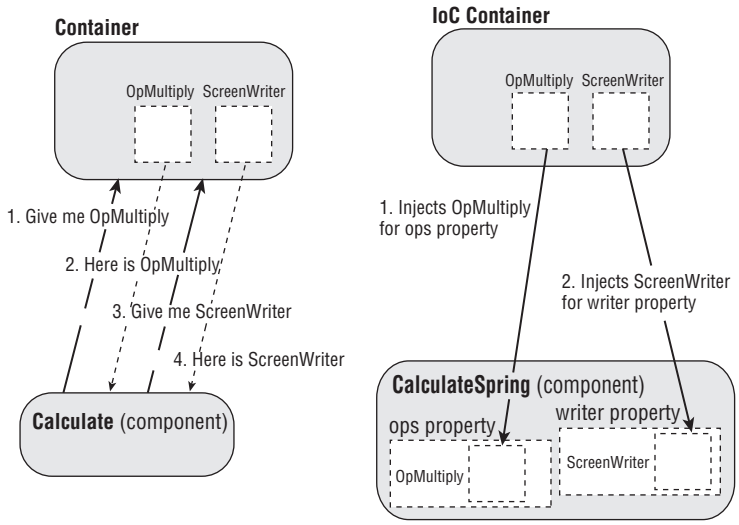

Figure 1-3

Dependency Injection

Since the Spring code shown is dependent on the availability of the DataSource instance to work, and the instance is injected into the component via the `setDs()` method, this technique is often referred to as *dependency injection*. In fact, the use of the setter method, `setDS()`, to inject the DataSource code dependency is known as *setter injection*. Spring also supports *constructor injection*, in which a dependent resource is injected into a bean via its constructor.

Try It Out Creating a Dependency Injection

To see dependency injection in action, you do not even have to rebuild your project — since beans are wired via the context descriptor in Spring.

1. Go back to the `src/chapter1/springfirst` directory and run `CalculateSpring`:

```
mvn exec:java -Dexec.mainClass=com.wrox.begspring.CalculateSpring -Dexec.args="3000 3"
```

   The current bean is autowired with an `Operation` implementation that adds two numbers together. And you see the output:

```
The result of 3000 plus 3 is 3003!
```

2. Change the directory to edit the context descriptor in the `src/chapter1/springfirst/target/classes` directory:

```
cd src/chatper1/springfirst/target/classes
```

3. Now, modify `beans.xml` to use the `OpMultiply` implementation instead: edit the file to match the highlighted lines shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

<bean id="screen" class="com.wrox.begspring.ScreenWriter" />
<bean id="multiply" class="com.wrox.begspring.OpMultiply" />
<bean id="add" class="com.wrox.begspring.OpAdd" />

<bean id="opsbean" class="com.wrox.begspring.CalculateSpring">
  <property name="ops" ref="multiply" />
  <property name="writer" ref="screen"/>
</bean>
</beans>
```

4. Change the directory back to the `/springfirst` directory, where `pom.xml` is located, and try running the command again:

```
mvn exec:java -Dexec.mainClass=com.wrox.begspring.CalculateSpring -Dexec.args="3000 3"
```

This time, the output is as follows:

```
The result of 3000 times 3 is 9000!
```

The application's behavior has been altered, and a different component injected into the setter, without any code recompilation.

How It Works

The `CalculateSpring` component code has a setter for the `ops` property, enabling the Spring IoC container to inject the dependency into the component during runtime. The following code shows the `setter` method in the highlighted portion:

```
package com.wrox.begspring;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class CalculateSpring {
  private Operation ops;
  private ResultWriter wtr;

  public void setOps(Operation ops) {
   this.ops = ops;
  }
  public  void setWriter(ResultWriter writer) {
   wtr = writer;
  }
```

When the `CalculateSpring` bean is wired in `beans.xml`, the setter dependency injection is used to wire an implementation of `OpMultiply`, instead of the former `OpAdd`, to this `ops` property. This setter injection is highlighted in the following code:

```
<bean id="opsbean" class="com.wrox.begspring.CalculateSpring">
  <property name="ops" ref="multiply" />
  <property name="writer" ref="screen"/>
</bean>
```

As you can see, dependency injection enables JavaBean components to be rewired without compilation in the IoC container. Decoupling the dependency from the JavaBean that depends on it, and deferring the selection until deployment or runtime, greatly enhances code reusability for the wired software component.

Adding Aspect-Oriented Programming to the Mix

Spring supports AOP, or aspect-oriented programming. AOP allows you to systematically apply a set of code modules, called *aspects,* to another (typically larger) body of target code. The end result is a shuffling of the aspect code with the body of code that it cuts across. However, the crosscutting aspects are coded and maintained separately, and the target code can be coded and maintained completely free of the crosscutting aspects. In AOP lingo, this is called *separation of concerns*.

The technique may appear strange initially. However, in most large software systems, a lot of code addresses a specific concern that can cut across many different source modules. These are called *crosscutting concerns* in AOP. Some typical examples of crosscutting concerns include security code, transaction code, and logging code. Traditional coding techniques do not allow such code to be created and maintained separately; instead it must be intermixed with other application logic.

As you can imagine, a code body that includes a mix of crosscutting concerns can be difficult to maintain. If you need to modify the crosscutting concern code, you may need to make changes across many files. In addition, crosscutting concern code tends to clutter up the application logic flow (for example, with security and logging code), making the code harder to understand.

In Figure 1-4, the set of logging aspects is applied to the main application using *pointcuts*. A pointcut in AOP describes where and how the aspect code should be inserted into the target. Figure 1-4 shows how Spring AOP matches pointcuts, and applies aspects to the target code at multiple matched join points.

Spring 2 supports AOP using two mechanisms. The first is via Spring AOP, and is a proxy-based implementation that existed in versions before 2.*x*. (A proxy in this sense is a Java wrapper class created or generated for the purpose of intercepting method invocations.) The second is via the integration with the AspectJ programming language and associated environments. AspectJ is one of the most popular AOP frameworks on the market today.

Chapter 12 focuses on Spring AOP and its use of AspectJ. Here you will get a small taste of what's to come.

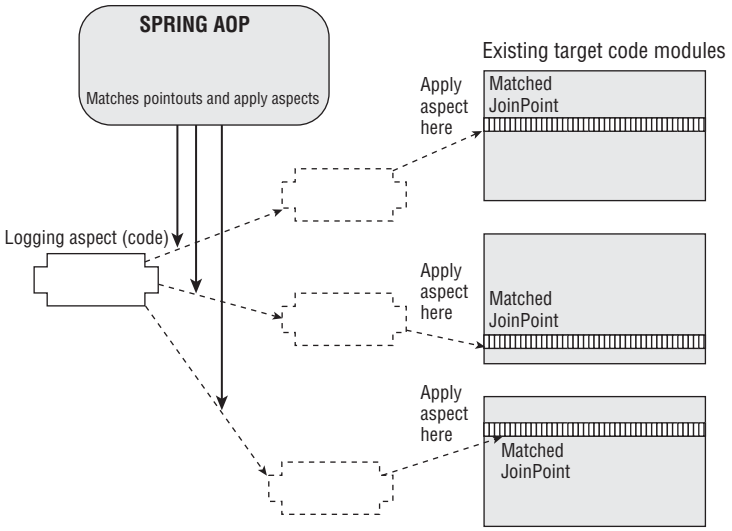

Figure 1-4

Adding a Logging Aspect

In this section, you can try out Spring AOP support by applying a logging aspect to the existing calculation code.

The code for the logging aspect is the `com.wrox.begspring.aspects.LoggingAspect` class, which you can find under the `src\chapter1\springaop\src\main\java\com\wrox\begspring\aspects` directory; it is shown here:

```
package com.wrox.begspring.aspects;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class LoggingAspect {

  @Before("execution(* com.wrox.begspring.Operation.*(..))")
  public void logMethodExecution(JoinPoint jp) {
    System.out.println("AOP logging -> "
        + jp.toShortString() );

  }
}
```

The logging aspect that can be applied is in boldface in the preceding code. Note that there is nothing about this aspect code that refers to the `CalculateSpring` code — the target to which it is applied. In fact, this aspect can be coded, modified, and maintained independently of the `CalculateSpring` code.

This code prints a logging message, similar to the following, whenever a method on the `Operation` interface (from the `CalculateSpring` code) is called:

```
AOP logging -> execution(getOpsName)
```

Try It Out Experimenting with Spring AOP Support

Once you have created an aspect, in this case a logging aspect, you can apply it to a target (in this case the `CalculateSpring` components) through the context descriptor. Try applying the aspect to the `CalcuateSpringAOP` class (just the good old `CalculateSpring` class renamed) by following these steps:

1. Change the directory to the `springaop` project in the source code distribution:

```
cd src/chapter1/springaop
```

2. Compile code using Maven 2, via the following command:

```
mvn compile
```

3. Run the code using Maven 2 by typing the following on one line:

```
mvn exec:java -Dexec.mainClass=com.wrox.begspring.CalculateSpringAOP
-Dexec.args="3000 3"
```

4. Take a look at the output; it should look like this:

```
AOP logging -> execution(getOpsName)
AOP logging -> execution(operate)
The result of 3000 times 3 is 9000!
```

Note that the executions of both the `getOpsName()` and `operate()` methods of the `Operation` interface are logged. Yet the implementations of this interface — such as `OpAdd` or `OpMultiply` — are not logged at all.

How It Works

The main code for `CalculateSpring` is renamed `CalculateSpringAOP`, and is shown here:

```
package com.wrox.begspring;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;


public class CalculateSpringAOP {

  private Operation ops;
  private ResultWriter wtr;
```

```
    public void setOps(Operation ops) {
     this.ops = ops;
    }
    public  void setWriter(ResultWriter writer) {
     wtr = writer;
    }

    public static void main(String[] args) {
      ApplicationContext context =
        new ClassPathXmlApplicationContext(
              "beans.xml");


      BeanFactory factory = (BeanFactory) context;
      CalculateSpringAOP calc =
        (CalculateSpringAOP) factory.getBean("opsbean");

      calc.execute(args);
    }



    public void execute(String [] args)  {
      long op1 = Long.parseLong(args[0]);
      long op2 = Long.parseLong(args[1]);
         wtr.showResult("The result of " + op1 +
             ops.getOpsName() + op2 + " is "
             + ops.operate(op1, op2) + "!");

    }

  }
```

Note that this main code has not been modified in any way. The aspect is applied via Spring AOP (underneath the covers it's using dynamic proxies) without requiring changes to the target body of code.

Wiring in AOP Proxies

As with other Spring techniques, the AOP magic is wired via the configuration in the Spring context descriptor. If you take a look at the `beans.xml` file used for this application (look in the `src/chapter1/springaop/src/main/resources` directory), you can see how the aspect is specified and applied. The following highlighted code is responsible for the application of the aspect:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
```

```
<bean id="screen" class="com.wrox.begspring.ScreenWriter" />
<bean id="multiply" class="com.wrox.begspring.OpMultiply" />


<bean id="opsbean" class="com.wrox.begspring.CalculateSpringAOP"
autowire="byType"/>

<aop:aspectj-autoproxy/>

<bean id="logaspect" class="com.wrox.begspring.aspects.LoggingAspect"/>

</beans>
```

The Spring container processes the `<aop:aspectj-autoproxy>` element and automatically creates a dynamic proxy required for AOP when it wires the beans together. Proxies are automatically added to the `CaculateSpringAOP` class by Spring, allowing for interception and the application of the `LoggingAspect`.

Note that you must add the highlighted AOP schema and namespace before the `<aop:aspectj-autoproxy>` element can work.

In this case, Spring 2 uses the AspectJ pointcut language (described in the next section) to determine where the proxies should be added for interception by the aspect.

Matching Join Points via AspectJ Pointcut Expressions

Java annotations are used to specify the pointcut to match when applying the aspect. The code responsible for this is highlighted here:

```
package com.wrox.begspring.aspects;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class LoggingAspect {

  @Before("execution(* com.wrox.begspring.Operation.*(..))")
  public void logMethodExecution(JoinPoint jp) {
    System.out.println("AOP logging -> "
        + jp.toShortString() );

  }
}
```

The `@Aspect` annotation marks this class as an Aspect. The `@Before()` annotation specifies the actual pointcut. The AspectJ pointcut expression language is used here to match a *join point* — a candidate spot in the target code where the aspect can be applied. This expression basically says, *Apply this aspect before you call any method on the type* `com.wrox.begspring.Operation`.

Since `com.wrox.begspring.Operation` is actually an interface, this results in aspect application before execution of any method on any implementation of this interface.

The argument to the `logMethodExecution()` method is a join point. This is a context object that marks the application point of the aspect. There are several methods on this join point that can be very useful in the aspect implementation. The following table describes some of the methods available at the join point.

Method	Description
`getArgs()`	Returns an object that can be used to access the argument of the join point
`getKind()`	Returns a string that describes the kind of join point precisely
`getSignature()`	Returns the method signature at this join point
`getSourceLocation()`	Returns information on the source code (filename, line, column) where this join point is matched
`getStaticPart()`	Returns an object that can be used to access the static (non–wild card) part of the join point
`getTarget()`	Returns the target object that the aspect is being applied to (at this join point)
`getThis()`	Returns the object that is currently executing
`toString()`	Returns the matched join point description as a string
`toShortString()`	Returns the matched join point description in a short format
`toLongString()`	Returns the matched join point description

Since an aspect is just a Java class, you can define as many as you need. You can imagine writing a set of logging aspects that can be maintained separately, and flexibly applied to a large body of code via manipulation of the pointcut expressions. This is the essence of how AOP can be very useful in everyday programming. Chapter 12 has a lot more coverage on Spring's AOP support.

Beyond Plumbing — Spring API Libraries

Thus far, you have seen that Spring provides:

- A way to wire JavaBeans together to create applications
- A framework that supports dependency injection
- A framework that supports AOP

These capabilities are fundamental to application plumbing and are independent of the type of software systems you are creating. Whether you are writing a simple small application that runs on a single machine and uses only file I/O, or a complex server-based application that services thousands of users, Spring can be very helpful. These capabilities can help make your design more modular, and your code more reusable.

They decouple dependency specifications from business logic, and separate crosscutting concerns. All these things are desirable in software projects of any type and/or size.

> *The Spring framework provides generic plumbing for creating modularized applications of any kind; you are not restricted to creating web-based enterprise Java applications.*

Going beyond the generic plumbing discussed so far, Spring offers much, much more.

Using Spring APIs To Facilitate Application Creation

Spring includes a large body of components and APIs that you can use when creating your application. Many of these components address specific type of applications, such as web-based server applications.

Figure 1-5 shows the distinct modules of APIs and components included with the Spring distribution.

You can see the Java EE–centric nature of Spring in Figure 1.5. All the API modules fall under one of the three main classifications: core container, Java EE support, or persistence.

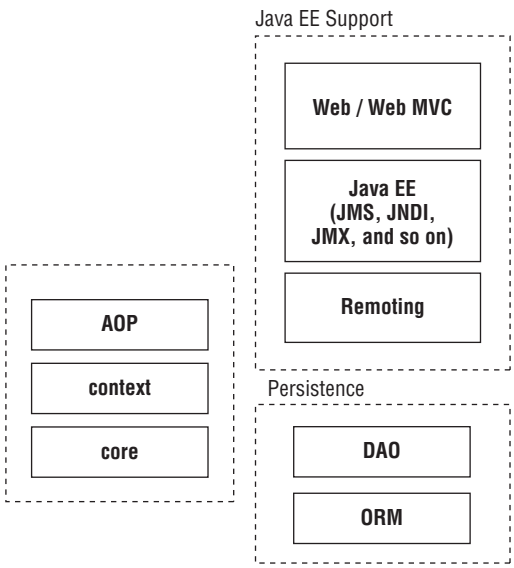

Figure 1-5

The following table describes the various categories of APIs that are available.

Spring 2 API Modules	Description
CORE	Lightweight Spring IoC container implementation, supporting dependency injection.
CONTEXT	A module that provides internationalization support, event handling, resource access, asynchronous processing, message triggered behaviors, and context management.

Spring 2 API Modules	Description
AOP	Supports AOP for application crosscutting concerns such as security, transactions, and other behaviors. Includes proxy-based implementations via Spring AOP, and integration with the AspectJ AOP language.
DAO	Provides a generic Data Access Objects abstraction over access of relational data from a variety of sources (such as JDBC). The access approach is uniform, regardless of the source of the data.
ORM	Provides uniform access to a wide variety of Object to Relational Mapping technologies including Hibernate, iBatis Maps, JDO, and JPA.
REMOTING	This API abstracts the ability to export interfaces for remote access to the features of a Spring components–based application. Supported access protocols include RMI, JMS, Hessian, Burlap, JAX RPC and Spring HTTP invoker. Remote access via Web Services is also supported through Spring Remoting.
WEB and WEB MVC	Components and APIs providing support for web applications, including requests handling, file uploads, portlet implementation, Struts and Webwork integration, and so on. Includes support for Model View-Controller (MVC) design patterns during the construction of web applications. Enables integration with a large variety of presentation technologies, including Velocity, JSP, JSF, and so on.
JEE	This API and its components support enterprise application creation, including the creation of those that support services found in the Java EE container — JMX, JMS, JDBC, JNDI, JTA, JPA, and so on. Also includes an API that supports the creation of standard EJBs using Spring components, and the ability to invoke EJBs from Spring components.

Bearing in mind that Spring was originally invented to create an easier and more lightweight alternative to the legacy J2EE, it is easy to see why such a rich API exists for component-based enterprise application development.

Throughout this book, the APIs and components depicted in Figure 1-5 are explored. You will have many hands-on opportunities to work and experiment with the Spring APIs.

Summary

Spring started life as an implementation of a lightweight alternative to the heavyweight J2EE 1.4 containers. It enables the construction of completely component-based applications with just enough components to carry out the work — and without the baggage of application servers.

The core of Spring provides flexible runtime wiring of Java Beans via an XML-based descriptor; applications can be created by wiring together JavaBeans components with Spring-supplied container-service components that supply essential services such as relational database access and user interface handling.

Inversion of control (IoC) is used throughout Spring applications to decouple dependencies that typically block reuse and increase complexity in J2EE environments. Instead of a component manually looking up the provider of a service, the provider is injected into the component by the Spring container at runtime. This enables you to write reusable components that are independent of provider-specific features.

AOP is a core enabler in Spring. AOP support enables you to factor crosscutting concerns out of your application and maintain them separately from the main body of code. Some common crosscutting concerns for applications include logging, security implementation, and transactions. The maintenance of such code in separate code modules called *aspects* makes the main body of code clearer and easier to maintain, while changes in the crosscutting concerns require only modification of the aspects, not of the (potentially large) main code body. Spring provides a proxy-based AOP implementation, as well as integration with the popular AspectJ AOP programming language.

In addition to the core application plumbing that can apply to any kind of Java application, Spring also provides a rich API and prefabricated component support for creating applications. These components and APIs can be used to build enterprise Java applications without the deployment of a conventional Java EE server.