The IoC Container

This chapter covers Spring's Inversion of Control (IoC) container.

Introduction to the Spring IoC Container and Beans

This chapter covers the Spring Framework implementation of the Inversion of Control (IoC) principle. IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies (that is, the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes or a mechanism such as the Service Locator pattern.

The `org.springframework.beans` and `org.springframework.context` packages are the basis for Spring Framework's IoC container. The {api-spring-framework}/beans/factory/BeanFactory.html[`BeanFactory`] interface provides an advanced configuration mechanism capable of managing any type of object. {api-spring-framework}/context/ApplicationContext.html[`ApplicationContext`] is a sub-interface of `BeanFactory`. It adds:

- Easier integration with Spring's AOP features
- Message resource handling (for use in internationalization)
- Event publication
- Application-layer specific contexts such as the `WebApplicationContext` for use in web applications.

In short, the `BeanFactory` provides the configuration framework and basic functionality, and the `ApplicationContext` adds more enterprise-specific functionality. The `ApplicationContext` is a complete superset of the `BeanFactory` and is used exclusively in this chapter in descriptions of Spring's IoC container. For more information on using the `BeanFactory` instead of the `ApplicationContext,` see The `BeanFactory`.

In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container. Otherwise, a bean is simply one of many objects in your application. Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.

Container Overview

The `org.springframework.context.ApplicationContext` interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich interdependencies between those objects.

Several implementations of the `ApplicationContext` interface are supplied with Spring. In stand-alone applications, it is common to create an instance of {api-spring-framework}/context/support/ClassPathXmlApplicationContext.html[`ClassPathXmlApplicationContext`] or {api-spring-framework}/context/support/FileSystemXmlApplicationContext.html[`FileSystemXmlApplicationContext`]. While XML has been the traditional format for defining configuration metadata, you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

In most application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a web application scenario, a simple eight (or so) lines of boilerplate web descriptor XML in the `web.xml` file of the application typically suffices (see Convenient ApplicationContext Instantiation for Web Applications). If you use the Spring Tool Suite (an Eclipse-powered development environment), you can easily create this boilerplate configuration with a few mouse clicks or keystrokes.

The following diagram shows a high-level view of how Spring works. Your application classes are combined with configuration metadata so that, after the `ApplicationContext` is created and initialized,you have a fully configured and executable system or application.

[container magic] | *images/container-magic.png*

Figure 1. The Spring IoC container

Configuration Metadata

As the preceding diagram shows, the Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.

Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container.

NOTE	XML-based metadata is not the only allowed form of configuration metadata. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. These days, many developers choose Java-based configuration for their Spring applications.

For information about using other forms of metadata with the Spring container, see: * Annotation-based configuration: Spring 2.5 introduced support for annotation-based configuration metadata. * Java-based configuration: Starting with Spring 3.0, many features provided by the Spring JavaConfig project became part of the core Spring Framework. Thus, you can define beans external to your application classes by using Java rather than XML files. To use these new features, see the `@Configuration`, `@Bean`, `@Import`, and `@DependsOn` annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. XML-based configuration metadata configures these beans as `<bean/>` elements inside a top-level `<beans/>` element. Java configuration typically uses `@Bean`-annotated methods within a `@Configuration` class.

These bean definitions correspond to the actual objects that make up your application. Typically, you define service layer objects, data access objects (DAOs), presentation objects such as Struts `Action` instances, infrastructure objects such as Hibernate `SessionFactories`, JMS `Queues`, and so forth. Typically, one does not configure fine-grained domain objects in the container, because it is usually the responsibility of DAOs and business logic to create and load domain objects. However, you can use Spring's integration with AspectJ to configure objects that have been created outside the control of an IoC container. See Using AspectJ to dependency-inject domain objects with Spring.

The following example shows the basic structure of XML-based configuration metadata:

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="..." class="..."> ① ②
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```

① The `id` attribute is a string that identifies the individual bean definition.

② The `class` attribute defines the type of the bean and uses the fully qualified classname.

The value of the `id` attribute refers to collaborating objects. The XML for referring to collaborating objects is not shown in this example. See Dependencies for more information.

Instantiating a Container

The location path or paths supplied to an `ApplicationContext` constructor are resource strings that let the container load configuration metadata from a variety of external resources, such as the local file system, the Java `CLASSPATH`, and so on.

```
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml",
"daos.xml");
```

NOTE After you learn about Spring's IoC container, you may want to know more about Spring's `Resource` abstraction (as described in [resources]), which provides a convenient mechanism for reading an InputStream from locations defined in a URI syntax. In particular, `Resource` paths are used to construct applications contexts, as described in [resources-app-ctx].

The following example shows the service layer objects `(services.xml)` configuration file:

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- services -->

    <bean id="petStore" class=
"org.springframework.samples.jpetstore.services.PetStoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions for services go here -->

</beans>
```

The following example shows the data access objects `daos.xml` file:

```xml
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="accountDao"
        class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <bean id="itemDao" class=
"org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions for data access objects go here -->

</beans>
```

In the preceding example, the service layer consists of the `PetStoreServiceImpl` class and two data access objects of the types `JpaAccountDao` and `JpaItemDao` (based on the JPA Object-Relational Mapping standard). The `property name` element refers to the name of the JavaBean property, and the `ref` element refers to the name of another bean definition. This linkage between `id` and `ref` elements expresses the dependency between collaborating objects. For details of configuring an object's dependencies, see Dependencies.

Composing XML-based Configuration Metadata

It can be useful to have bean definitions span multiple XML files. Often, each individual XML configuration file represents a logical layer or module in your architecture.

You can use the application context constructor to load bean definitions from all these XML fragments. This constructor takes multiple `Resource` locations, as was shown in the previous section. Alternatively, use one or more occurrences of the `<import/>` element to load bean definitions from another file or files. The following example shows how to do so:

```xml
<beans>
    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>
</beans>
```

In the preceding example, external bean definitions are loaded from three files: `services.xml`, `messageSource.xml`, and `themeSource.xml`. All location paths are relative to the definition file doing the importing, so `services.xml` must be in the same directory or classpath location as the file doing the importing, while `messageSource.xml` and `themeSource.xml` must be in a `resources` location below the location of the importing file. As you can see, a leading slash is ignored. However, given that these paths are relative, it is better form not to use the slash at all. The contents of the files being imported, including the top level `<beans/>` element, must be valid XML bean definitions, according to the Spring Schema.

> **NOTE**
>
> It is possible, but not recommended, to reference files in parent directories using a relative "../" path. Doing so creates a dependency on a file that is outside the current application. In particular, this reference is not recommended for `classpath:` URLs (for example, `classpath:../services.xml`), where the runtime resolution process chooses the "nearest" classpath root and then looks into its parent directory. Classpath configuration changes may lead to the choice of a different, incorrect directory.
>
> You can always use fully qualified resource locations instead of relative paths: for example, `file:C:/config/services.xml` or `classpath:/config/services.xml`. However, be aware that you are coupling your application's configuration to specific absolute locations. It is generally preferable to keep an indirection for such absolute locations — for example, through "${…}" placeholders that are resolved against JVM system properties at runtime.

The namespace itself provices the import directive feature. Further configuration features beyond plain bean definitions are available in a selection of XML namespaces provided by Spring — for example, the `context` and `util` namespaces.

The Groovy Bean Definition DSL

As a further example for externalized configuration metadata, bean definitions can also be expressed in Spring's Groovy Bean Definition DSL, as known from the Grails framework. Typically, such configuration live in a ".groovy" file with the structure shown in the following example:

```
beans {
    dataSource(BasicDataSource) {
        driverClassName = "org.hsqldb.jdbcDriver"
        url = "jdbc:hsqldb:mem:grailsDB"
        username = "sa"
        password = ""
        settings = [mynew:"setting"]
    }
    sessionFactory(SessionFactory) {
        dataSource = dataSource
    }
    myService(MyService) {
        nestedBean = { AnotherBean bean ->
            dataSource = dataSource
        }
    }
}
```

This configuration style is largely equivalent to XML bean definitions and even supports Spring's XML configuration namespaces. It also allows for importing XML bean definition files through an `importBeans` directive.

Using the Container

The `ApplicationContext` is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. By using the method `T getBean(String name, Class<T> requiredType)`, you can retrieve instances of your beans.

The `ApplicationContext` lets you read bean definitions and access them, as the following example shows:

```
// create and configure beans
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml",
"daos.xml");

// retrieve configured instance
PetStoreService service = context.getBean("petStore", PetStoreService.class);

// use configured instance
List<String> userList = service.getUsernameList();
```

With Groovy configuration, bootstrapping looks very similar. It has a different context implementation class which is Groovy-aware (but also understands XML bean definitions). The following example shows Groovy configuration:

```
ApplicationContext context = new GenericGroovyApplicationContext("services.groovy",
"daos.groovy");
```

The most flexible variant is `GenericApplicationContext` in combination with reader delegates — for example, with `XmlBeanDefinitionReader` for XML files, as the following example shows:

```
GenericApplicationContext context = new GenericApplicationContext();
new XmlBeanDefinitionReader(context).loadBeanDefinitions("services.xml", "
daos.xml");
context.refresh();
```

You can also use the `GroovyBeanDefinitionReader` for Groovy files, as the following example shows:

```
GenericApplicationContext context = new GenericApplicationContext();
new GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy",
"daos.groovy");
context.refresh();
```

You can mix and match such reader delegates on the same `ApplicationContext`, reading bean definitions from diverse configuration sources.

You can then use `getBean` to retrieve instances of your beans. The `ApplicationContext` interface has a few other methods for retrieving beans, but, ideally, your application code should never use them. Indeed, your application code should have no calls to the `getBean()` method at all and thus have no dependency on Spring APIs at all. For example, Spring's integration with web frameworks provides dependency injection for various web framework components such as controllers and JSF-managed beans, letting you declare a dependency on a specific bean through metadata (such as an autowiring annotation).

Bean Overview

A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container (for example, in the form of XML `<bean/>` definitions).

Within the container itself, these bean definitions are represented as `BeanDefinition` objects, which contain (among other information) the following metadata:

- A package-qualified class name: typically, the actual implementation class of the bean being defined.
- Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle callbacks, and so forth).
- References to other beans that are needed for the bean to do its work. These references are also

called collaborators or dependencies.

- Other configuration settings to set in the newly created object — for example, the size limit of the pool or the number of connections to use in a bean that manages a connection pool.

This metadata translates to a set of properties that make up each bean definition. The following table describes these properties:

Table 1. The bean definition

Property	Explained in...
Class	Instantiating Beans
Name	Naming Beans
Scope	Bean Scopes
Constructor arguments	Dependency Injection
Properties	Dependency Injection
Autowiring mode	Autowiring Collaborators
Lazy initialization mode	Lazy-initialized Beans
Initialization method	Initialization Callbacks
Destruction method	Destruction Callbacks

In addition to bean definitions that contain information on how to create a specific bean, the `ApplicationContext` implementations also permit the registration of existing objects that are created outside the container (by users). This is done by accessing the ApplicationContext's BeanFactory through the `getBeanFactory()` method, which returns the BeanFactory `DefaultListableBeanFactory` implementation. `DefaultListableBeanFactory` supports this registration through the `registerSingleton(..)` and `registerBeanDefinition(..)` methods. However, typical applications work solely with beans defined through regular bean definition metadata.

NOTE Bean metadata and manually supplied singleton instances need to be registered as early as possible, in order for the container to properly reason about them during autowiring and other introspection steps. While overriding existing metadata and existing singleton instances is supported to some degree, the registration of new beans at runtime (concurrently with live access to the factory) is not officially supported and may lead to concurrent access exceptions, inconsistent state in the bean container, or both.

Naming Beans

Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean. A bean usually has only one identifier. However, if it requires more than one, the extra ones can be considered aliases.

In XML-based configuration metadata, you use the `id` attribute, the `name` attribute, or both to specify the bean identifiers. The `id` attribute lets you specify exactly one id. Conventionally, these names are alphanumeric ('myBean', 'someService', etc.), but they can contain special characters as well. If you want to introduce other aliases for the bean, you can also specify them in the `name` attribute, separated by a comma (`,`), semicolon (`;`), or white space. As a historical note, in versions prior to Spring 3.1, the `id` attribute was defined as an `xsd:ID` type, which constrained possible characters. As of 3.1, it is defined as an `xsd:string` type. Note that bean `id` uniqueness is still enforced by the container, though no longer by XML parsers.

You are not required to supply a `name` or an `id` for a bean. If you do not supply a `name` or `id` explicitly, the container generates a unique name for that bean. However, if you want to refer to that bean by name, through the use of the `ref` element or a Service Locator style lookup, you must provide a name. Motivations for not supplying a name are related to using inner beans and autowiring collaborators.

Bean Naming Conventions

The convention is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter and are camel-cased from there. Examples of such names include `accountManager`, `accountService`, `userDao`, `loginController`, and so forth.

Naming beans consistently makes your configuration easier to read and understand. Also, if you use Spring AOP, it helps a lot when applying advice to a set of beans related by name.

NOTE With component scanning in the classpath, Spring generates bean names for unnamed components, following the rules described earlier: essentially, taking the simple class name and turning its initial character to lower-case. However, in the (unusual) special case when there is more than one character and both the first and second characters are upper case, the original casing gets preserved. These are the same rules as defined by `java.beans.Introspector.decapitalize` (which Spring uses here).

Aliasing a Bean outside the Bean Definition

In a bean definition itself, you can supply more than one name for the bean, by using a combination of up to one name specified by the `id` attribute and any number of other names in the `name` attribute. These names can be equivalent aliases to the same bean and are useful for some situations, such as letting each component in an application refer to a common dependency by using a bean name that is specific to that component itself.

Specifying all aliases where the bean is actually defined is not always adequate, however. It is sometimes desirable to introduce an alias for a bean that is defined elsewhere. This is commonly the case in large systems where configuration is split amongst each subsystem, with each subsystem having its own set of object definitions. In XML-based configuration metadata, you can use the `<alias/>` element to accomplish this. The following example shows how to do so:

```xml
<alias name="fromName" alias="toName"/>
```

In this case, a bean (in the same container) named `fromName` may also, after the use of this alias definition, be referred to as `toName`.

For example, the configuration metadata for subsystem A may refer to a DataSource by the name of `subsystemA-dataSource`. The configuration metadata for subsystem B may refer to a DataSource by the name of `subsystemB-dataSource`. When composing the main application that uses both these subsystems, the main application refers to the DataSource by the name of `myApp-dataSource`. To have all three names refer to the same object, you can add the following alias definitions to the configuration metadata:

```xml
<alias name="myApp-dataSource" alias="subsystemA-dataSource"/>
<alias name="myApp-dataSource" alias="subsystemB-dataSource"/>
```

Now each component and the main application can refer to the dataSource through a name that is unique and guaranteed not to clash with any other definition (effectively creating a namespace), yet they refer to the same bean.

Java-configuration

If you use Javaconfiguration, the `@Bean` annotation can be used to provide aliases. See Using the `@Bean` Annotation for details.

Instantiating Beans

A bean definition is essentially a recipe for creating one or more objects. The container looks at the recipe for a named bean when asked and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object.

If you use XML-based configuration metadata, you specify the type (or class) of object that is to be instantiated in the `class` attribute of the `<bean/>` element. This `class` attribute (which, internally, is a `Class` property on a `BeanDefinition` instance) is usually mandatory. (For exceptions, see Instantiation by Using an Instance Factory Method and Bean Definition Inheritance.) You can use the `Class` property in one of two ways:

- Typically, to specify the bean class to be constructed in the case where the container itself directly creates the bean by calling its constructor reflectively, somewhat equivalent to Java code with the `new` operator.
- To specify the actual class containing the `static` factory method that is invoked to create the object, in the less common case where the container invokes a `static` factory method on a class to create the bean. The object type returned from the invocation of the `static` factory method

may be the same class or another class entirely.

> *Inner class names*
>
> If you want to configure a bean definition for a `static` nested class, you have to use the binary name of the nested class.
>
> For example, if you have a class called `SomeThing` in the `com.example` package, and this `SomeThing` class has a `static` nested class called `OtherThing`, the value of the `class` attribute on a bean definition would be `com.example.SomeThing$OtherThing`.
>
> Notice the use of the `$` character in the name to separate the nested class name from the outer class name.

Instantiation with a Constructor

When you create a bean by the constructor approach, all normal classes are usable by and compatible with Spring. That is, the class being developed does not need to implement any specific interfaces or to be coded in a specific fashion. Simply specifying the bean class should suffice. However, depending on what type of IoC you use for that specific bean, you may need a default (empty) constructor.

The Spring IoC container can manage virtually any class you want it to manage. It is not limited to managing true JavaBeans. Most Spring users prefer actual JavaBeans with only a default (no-argument) constructor and appropriate setters and getters modeled after the properties in the container. You can also have more exotic non-bean-style classes in your container. If, for example, you need to use a legacy connection pool that absolutely does not adhere to the JavaBean specification, Spring can manage it as well.

With XML-based configuration metadata you can specify your bean class as follows:

```xml
<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

For details about the mechanism for supplying arguments to the constructor (if required) and setting object instance properties after the object is constructed, see Injecting Dependencies.

Instantiation with a Static Factory Method

When defining a bean that you create with a static factory method, use the `class` attribute to specify the class that contains the `static` factory method and an attribute named `factory-method` to specify the name of the factory method itself. You should be able to call this method (with optional arguments, as described later) and return a live object, which subsequently is treated as if it had been created through a constructor. One use for such a bean definition is to call `static` factories in legacy code.

The following bean definition specifies that the bean be created by calling a factory method. The definition does not specify the type (class) of the returned object, only the class containing the factory method. In this example, the `createInstance()` method must be a static method. The following example shows how to specify a factory method:

```
<bean id="clientService"
    class="examples.ClientService"
    factory-method="createInstance"/>
```

The following example shows a class that would work with the preceding bean definition:

```
public class ClientService {
    private static ClientService clientService = new ClientService();
    private ClientService() {}

    public static ClientService createInstance() {
        return clientService;
    }
}
```

For details about the mechanism for supplying (optional) arguments to the factory method and setting object instance properties after the object is returned from the factory, see Dependencies and Configuration in Detail.

Instantiation by Using an Instance Factory Method

Similar to instantiation through a static factory method, instantiation with an instance factory method invokes a non-static method of an existing bean from the container to create a new bean. To use this mechanism, leave the `class` attribute empty and, in the `factory-bean` attribute, specify the name of a bean in the current (or parent or ancestor) container that contains the instance method that is to be invoked to create the object. Set the name of the factory method itself with the `factory-method` attribute. The following example shows how to configure such a bean:

```xml
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- inject any dependencies required by this locator bean -->
</bean>

<!-- the bean to be created via the factory bean -->
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
```

The following example shows the corresponding Java class:

```java
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();

    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

One factory class can also hold more than one factory method, as the following example shows:

```xml
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- inject any dependencies required by this locator bean -->
</bean>

<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>

<bean id="accountService"
    factory-bean="serviceLocator"
    factory-method="createAccountServiceInstance"/>
```

The following example shows the corresponding Java class:

```java
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();

    private static AccountService accountService = new AccountServiceImpl();

    public ClientService createClientServiceInstance() {
        return clientService;
    }

    public AccountService createAccountServiceInstance() {
        return accountService;
    }
}
```

This approach shows that the factory bean itself can be managed and configured through dependency injection (DI). See Dependencies and Configuration in Detail.

NOTE In Spring documentation, “factory bean” refers to a bean that is configured in the Spring container and that creates objects through an instance or static factory method. By contrast, `FactoryBean` (notice the capitalization) refers to a Spring-specific `FactoryBean` .

Dependencies

A typical enterprise application does not consist of a single object (or bean in the Spring parlance). Even the simplest application has a few objects that work together to present what the end-user sees as a coherent application. This next section explains how you go from defining a number of bean definitions that stand alone to a fully realized application where objects collaborate to achieve a goal.

Dependency Injection

Dependency injection (DI) is a process whereby objects define their dependencies (that is, the other objects with which they work) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method. The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies on its own by using direct construction of classes or the Service Locator pattern.

Code is cleaner with the DI principle, and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies and does not know the location or class of the dependencies. As a result, your classes become easier to test, particularly when the dependencies are on interfaces or abstract base classes, which allow for stub or mock implementations to be used in unit tests.

DI exists in two major variants: Constructor-based dependency injection and Setter-based dependency injection.

Constructor-based Dependency Injection

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency. Calling a `static` factory method with specific arguments to construct the bean is nearly equivalent, and this discussion treats arguments to a constructor and to a `static` factory method similarly. The following example shows a class that can only be dependency-injected with constructor injection:

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on a MovieFinder
    private MovieFinder movieFinder;

    // a constructor so that the Spring container can inject a MovieFinder
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually uses the injected MovieFinder is omitted...
}
```

Notice that there is nothing special about this class. It is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

Constructor Argument Resolution

Constructor argument resolution matching occurs by using the argument’s type. If no potential ambiguity exists in the constructor arguments of a bean definition, the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor when the bean is being instantiated. Consider the following class:

```java
package x.y;

public class ThingOne {

    public ThingOne(ThingTwo thingTwo, ThingThree thingThree) {
        // ...
    }
}
```

Assuming that ThingTwo and ThingThree classes are not related by inheritance, no potential ambiguity exists. Thus, the following configuration works fine, and you do not need to specify the constructor argument indexes or types explicitly in the <constructor-arg/> element.

```xml
<beans>
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg ref="beanTwo"/>
        <constructor-arg ref="beanThree"/>
    </bean>

    <bean id="beanTwo" class="x.y.ThingTwo"/>

    <bean id="beanThree" class="x.y.ThingThree"/>
</beans>
```

When another bean is referenced, the type is known, and matching can occur (as was the case with the preceding example). When a simple type is used, such as <value>true</value>, Spring cannot determine the type of the value, and so cannot match by type without help. Consider the following class:

```java
package examples;

public class ExampleBean {

    // Number of years to calculate the Ultimate Answer
    private int years;

    // The Answer to Life, the Universe, and Everything
    private String ultimateAnswer;

    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

Constructor argument type matching

In the preceding scenario, the container can use type matching with simple types if you explicitly specify the type of the constructor argument by using the `type` attribute. as the following example shows:

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg type="int" value="7500000"/>
    <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

Constructor argument index

You can use the `index` attribute to specify explicitly the index of constructor arguments, as the following example shows:

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg index="0" value="7500000"/>
    <constructor-arg index="1" value="42"/>
</bean>
```

In addition to resolving the ambiguity of multiple simple values, specifying an index resolves ambiguity where a constructor has two arguments of the same type.

NOTE The index is 0-based.

Constructor argument name

You can also use the constructor parameter name for value disambiguation, as the following example shows:

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg name="years" value="7500000"/>
    <constructor-arg name="ultimateAnswer" value="42"/>
</bean>
```

Keep in mind that, to make this work out of the box, your code must be compiled with the debug flag enabled so that Spring can look up the parameter name from the constructor. If you cannot or do not want to compile your code with the debug flag, you can use the @ConstructorProperties JDK annotation to explicitly name your constructor arguments. The sample class would then have to look as follows:

```java
package examples;

public class ExampleBean {

    // Fields omitted

    @ConstructorProperties({"years", "ultimateAnswer"})
    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

Setter-based Dependency Injection

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or a no-argument `static` factory method to instantiate your bean.

The following example shows a class that can only be dependency-injected by using pure setter injection. This class is conventional Java. It is a POJO that has no dependencies on container specific interfaces, base classes, or annotations.

```java
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;

    // a setter method so that the Spring container can inject a MovieFinder
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually uses the injected MovieFinder is omitted...
}
```

The `ApplicationContext` supports constructor-based and setter-based DI for the beans it manages. It also supports setter-based DI after some dependencies have already been injected through the constructor approach. You configure the dependencies in the form of a `BeanDefinition`, which you use in conjunction with `PropertyEditor` instances to convert properties from one format to another. However, most Spring users do not work with these classes directly (that is, programmatically) but rather with XML `bean` definitions, annotated components (that is, classes annotated with `@Component`, `@Controller`, and so forth), or `@Bean` methods in Java-based `@Configuration` classes. These sources are then converted internally into instances of `BeanDefinition` and used to load an entire Spring IoC container instance.

Constructor-based or setter-based DI?

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies. Note that use of the @Required annotation on a setter method can be used to make the property be a required dependency; however, constructor injection with programmatic validation of arguments is preferable.

The Spring team generally advocates constructor injection, as it lets you implement application components as immutable objects and ensures that required dependencies are not `null`. Furthermore, constructor-injected components are always returned to the client (calling) code in a fully initialized state. As a side note, a large number of constructor arguments is a bad code smell, implying that the class likely has too many responsibilities and should be refactored to better address proper separation of concerns.

Setter injection should primarily only be used for optional dependencies that can be assigned reasonable default values within the class. Otherwise, not-null checks must be performed everywhere the code uses the dependency. One benefit of setter injection is that setter methods make objects of that class amenable to reconfiguration or re-injection later. Management through JMX MBeans is therefore a compelling use case for setter injection.

Use the DI style that makes the most sense for a particular class. Sometimes, when dealing with third-party classes for which you do not have the source, the choice is made for you. For example, if a third-party class does not expose any setter methods, then constructor injection may be the only available form of DI.

Dependency Resolution Process

The container performs bean dependency resolution as follows:

- The `ApplicationContext` is created and initialized with configuration metadata that describes all the beans. Configuration metadata can be specified by XML, Java code, or annotations.
- For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method (if you use that instead of a normal constructor). These dependencies are provided to the bean, when the bean is actually created.
- Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.
- Each property or constructor argument that is a value is converted from its specified format to the actual type of that property or constructor argument. By default, Spring can convert a value supplied in string format to all built-in types, such as `int`, `long`, `String`, `boolean`, and so forth.

The Spring container validates the configuration of each bean as the container is created. However, the bean properties themselves are not set until the bean is actually created. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created. Scopes are defined in Bean Scopes. Otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean’s dependencies and its dependencies' dependencies (and so on) are created and assigned. Note that resolution mismatches among those dependencies may show up late — that is, on first creation of the affected bean.

Circular dependencies

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a `BeanCurrentlyInCreationException`.

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.

Unlike the typical case (with no circular dependencies), a circular dependency between bean A and bean B forces one of the beans to be injected into the other prior to being fully initialized itself (a classic chicken-and-egg scenario).

You can generally trust Spring to do the right thing. It detects configuration problems, such as references to non-existent beans and circular dependencies, at container load-time. Spring sets properties and resolves dependencies as late as possible, when the bean is actually created. This means that a Spring container that has loaded correctly can later generate an exception when you request an object if there is a problem creating that object or one of its dependencies — for example, the bean throws an exception as a result of a missing or invalid property. This potentially delayed visibility of some configuration issues is why `ApplicationContext` implementations by default pre-instantiate singleton beans. At the cost of some upfront time and memory to create these beans before they are actually needed, you discover configuration issues when the `ApplicationContext` is created, not later. You can still override this default behavior so that singleton beans initialize lazily, rather than being pre-instantiated.

If no circular dependencies exist, when one or more collaborating beans are being injected into a dependent bean, each collaborating bean is totally configured prior to being injected into the dependent bean. This means that, if bean A has a dependency on bean B, the Spring IoC container completely configures bean B prior to invoking the setter method on bean A. In other words, the bean is instantiated (if it is not a pre-instantiated singleton), its dependencies are set, and the relevant lifecycle methods (such as a configured init method or the InitializingBean callback method) are invoked.

Examples of Dependency Injection

The following example uses XML-based configuration metadata for setter-based DI. A small part of a Spring XML configuration file specifies some bean definitions as follows:

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- setter injection using the nested ref element -->
    <property name="beanOne">
        <ref bean="anotherExampleBean"/>
    </property>

    <!-- setter injection using the neater ref attribute -->
    <property name="beanTwo" ref="yetAnotherBean"/>
    <property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

The following example shows the corresponding `ExampleBean` class:

```java
public class ExampleBean {

    private AnotherBean beanOne;

    private YetAnotherBean beanTwo;

    private int i;

    public void setBeanOne(AnotherBean beanOne) {
        this.beanOne = beanOne;
    }

    public void setBeanTwo(YetAnotherBean beanTwo) {
        this.beanTwo = beanTwo;
    }

    public void setIntegerProperty(int i) {
        this.i = i;
    }
}
```

In the preceding example, setters are declared to match against the properties specified in the XML file. The following example uses constructor-based DI:

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- constructor injection using the nested ref element -->
    <constructor-arg>
        <ref bean="anotherExampleBean"/>
    </constructor-arg>

    <!-- constructor injection using the neater ref attribute -->
    <constructor-arg ref="yetAnotherBean"/>

    <constructor-arg type="int" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

The following example shows the corresponding `ExampleBean` class:

```java
public class ExampleBean {

    private AnotherBean beanOne;

    private YetAnotherBean beanTwo;

    private int i;

    public ExampleBean(
        AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
        this.beanOne = anotherBean;
        this.beanTwo = yetAnotherBean;
        this.i = i;
    }
}
```

The constructor arguments specified in the bean definition are used as arguments to the constructor of the `ExampleBean`.

Now consider a variant of this example, where, instead of using a constructor, Spring is told to call a `static` factory method to return an instance of the object:

```xml
<bean id="exampleBean" class="examples.ExampleBean" factory-method="
createInstance">
    <constructor-arg ref="anotherExampleBean"/>
    <constructor-arg ref="yetAnotherBean"/>
    <constructor-arg value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

The following example shows the corresponding `ExampleBean` class:

```java
public class ExampleBean {

    // a private constructor
    private ExampleBean(...) {
        ...
    }

    // a static factory method; the arguments to this method can be
    // considered the dependencies of the bean that is returned,
    // regardless of how those arguments are actually used.
    public static ExampleBean createInstance (
        AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {

        ExampleBean eb = new ExampleBean (...);
        // some other operations...
        return eb;
    }
}
```

Arguments to the `static` factory method are supplied by `<constructor-arg/>` elements, exactly the same as if a constructor had actually been used. The type of the class being returned by the factory method does not have to be of the same type as the class that contains the `static` factory method (although, in this example, it is). An instance (non-static) factory method can be used in an essentially identical fashion (aside from the use of the `factory-bean` attribute instead of the `class` attribute), so we do not discuss those details here.

Dependencies and Configuration in Detail

As mentioned in the previous section, you can define bean properties and constructor arguments as references to other managed beans (collaborators) or as values defined inline. Spring's XML-based configuration metadata supports sub-element types within its `<property/>` and `<constructor-arg/>` elements for this purpose.

Straight Values (Primitives, Strings, and so on)

The `value` attribute of the `<property/>` element specifies a property or constructor argument as a human-readable string representation. Spring's conversion service is used to convert these values from a `String` to the actual type of the property or argument. The following example shows various values being set:

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
    <!-- results in a setDriverClassName(String) call -->
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
    <property name="username" value="root"/>
    <property name="password" value="masterkaoli"/>
</bean>
```

The following example uses the p-namespace for even more succinct XML configuration:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="com.mysql.jdbc.Driver"
        p:url="jdbc:mysql://localhost:3306/mydb"
        p:username="root"
        p:password="masterkaoli"/>

</beans>
```

The preceding XML is more succinct. However, typos are discovered at runtime rather than design time, unless you use an IDE (such as IntelliJ IDEA or the Spring Tool Suite) that supports automatic property completion when you create bean definitions. Such IDE assistance is highly recommended.

You can also configure a `java.util.Properties` instance, as follows:

```xml
<bean id="mappings"
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer
">

    <!-- typed as a java.util.Properties -->
    <property name="properties">
        <value>
            jdbc.driver.className=com.mysql.jdbc.Driver
            jdbc.url=jdbc:mysql://localhost:3306/mydb
        </value>
    </property>
</bean>
```

The Spring container converts the text inside the `<value/>` element into a `java.util.Properties` instance by using the JavaBeans `PropertyEditor` mechanism. This is a nice shortcut, and is one of a few places where the Spring team do favor the use of the nested `<value/>` element over the `value` attribute style.

The `idref` element

The `idref` element is simply an error-proof way to pass the `id` (a string value - not a reference) of another bean in the container to a `<constructor-arg/>` or `<property/>` element. The following example shows how to use it:

```xml
<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">
    <property name="targetName">
        <idref bean="theTargetBean"/>
    </property>
</bean>
```

The preceding bean definition snippet is exactly equivalent (at runtime) to the following snippet:

```xml
<bean id="theTargetBean" class="..." />

<bean id="client" class="...">
    <property name="targetName" value="theTargetBean"/>
</bean>
```

The first form is preferable to the second, because using the `idref` tag lets the container validate at deployment time that the referenced, named bean actually exists. In the second variation, no validation is performed on the value that is passed to the `targetName` property of the `client` bean. Typos are only discovered (with most likely fatal results) when the `client` bean is actually instantiated. If the `client` bean is a prototype bean, this typo and the resulting exception may only be discovered long after the container is deployed.

NOTE	The `local` attribute on the `idref` element is no longer supported in the 4.0 beans XSD, since it does not provide value over a regular `bean` reference any more. Change your existing `idref local` references to `idref bean` when upgrading to the 4.0 schema.

A common place (at least in versions earlier than Spring 2.0) where the `<idref/>` element brings value is in the configuration of AOP interceptors in a `ProxyFactoryBean` bean definition. Using `<idref/>` elements when you specify the interceptor names prevents you from misspelling an interceptor ID.

References to Other Beans (Collaborators)

The `ref` element is the final element inside a `<constructor-arg/>` or `<property/>` definition element. Here, you set the value of the specified property of a bean to be a reference to another bean (a collaborator) managed by the container. The referenced bean is a dependency of the bean whose property is to be set, and it is initialized on demand as needed before the property is set. (If the collaborator is a singleton bean, it may already be initialized by the container.) All references are ultimately a reference to another object. Scoping and validation depend on whether you specify the ID or name of the other object through the `bean`, `local,` or `parent` attributes.

Specifying the target bean through the `bean` attribute of the `<ref/>` tag is the most general form and allows creation of a reference to any bean in the same container or parent container, regardless of whether it is in the same XML file. The value of the `bean` attribute may be the same as the `id` attribute of the target bean or be the same as one of the values in the `name` attribute of the target bean. The following example shows how to use a `ref` element:

```
<ref bean="someBean"/>
```

Specifying the target bean through the `parent` attribute creates a reference to a bean that is in a parent container of the current container. The value of the `parent` attribute may be the same as either the `id` attribute of the target bean or one of the values in the `name` attribute of the target bean. The target bean must be in a parent container of the current one. You should use this bean reference variant mainly when you have a hierarchy of containers and you want to wrap an existing bean in a parent container with a proxy that has the same name as the parent bean. The following pair of listings shows how to use the `parent` attribute:

```
<!-- in the parent context -->
<bean id="accountService" class="com.something.SimpleAccountService">
    <!-- insert dependencies as required as here -->
</bean>
```

```
<!-- in the child (descendant) context -->
<bean id="accountService" <!-- bean name is the same as the parent bean -->
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref parent="accountService"/> <!-- notice how we refer to the parent bean
-->
    </property>
    <!-- insert other configuration and dependencies as required here -->
</bean>
```

NOTE The `local` attribute on the `ref` element is no longer supported in the 4.0 beans XSD, since it does not provide value over a regular `bean` reference any more. Change your existing `ref local` references to `ref bean` when upgrading to the 4.0 schema.

Inner Beans

A `<bean/>` element inside the `<property/>` or `<constructor-arg/>` elements defines an inner bean, as the following example shows:

```
<bean id="outer" class="...">
    <!-- instead of using a reference to a target bean, simply define the target
bean inline -->
    <property name="target">
        <bean class="com.example.Person"> <!-- this is the inner bean -->
            <property name="name" value="Fiona Apple"/>
            <property name="age" value="25"/>
        </bean>
    </property>
</bean>
```

An inner bean definition does not require a defined ID or name. If specified, the container does not use such a value as an identifier. The container also ignores the `scope` flag on creation, because inner beans are always anonymous and are always created with the outer bean. It is not possible to access inner beans independently or to inject them into collaborating beans other than into the enclosing bean.

As a corner case, it is possible to receive destruction callbacks from a custom scope — for example, for a request-scoped inner bean contained within a singleton bean. The creation of the inner bean instance is tied to its containing bean, but destruction callbacks let it participate in the request scope's lifecycle. This is not a common scenario. Inner beans typically simply share their containing bean's scope.

Collections

The `<list/>`, `<set/>`, `<map/>`, and `<props/>` elements set the properties and arguments of the Java `Collection` types `List`, `Set`, `Map`, and `Properties`, respectively. The following example shows how to use them:

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.org</prop>
            <prop key="support">support@example.org</prop>
            <prop key="development">development@example.org</prop>
        </props>
    </property>
    <!-- results in a setSomeList(java.util.List) call -->
    <property name="someList">
        <list>
            <value>a list element followed by a reference</value>
            <ref bean="myDataSource" />
        </list>
    </property>
    <!-- results in a setSomeMap(java.util.Map) call -->
    <property name="someMap">
        <map>
            <entry key="an entry" value="just some string"/>
            <entry key ="a ref" value-ref="myDataSource"/>
        </map>
    </property>
    <!-- results in a setSomeSet(java.util.Set) call -->
    <property name="someSet">
        <set>
            <value>just some string</value>
            <ref bean="myDataSource" />
        </set>
    </property>
</bean>
```

The value of a map key or value, or a set value, can also be any of the following elements:

```
bean | ref | idref | list | set | map | props | value | null
```

Collection Merging

The Spring container also supports merging collections. An application developer can define a parent `<list/>`, `<map/>`, `<set/>` or `<props/>` element and have child `<list/>`, `<map/>`, `<set/>` or `<props/>` elements inherit and override values from the parent collection. That is, the child collection's values are the result of merging the elements of the parent and child collections, with the child's collection elements overriding values specified in the parent collection.

This section on merging discusses the parent-child bean mechanism. Readers unfamiliar with parent and child bean definitions may wish to read the relevant section before continuing.

The following example demonstrates collection merging:

```
<beans>
    <bean id="parent" abstract="true" class="example.ComplexObject">
        <property name="adminEmails">
            <props>
                <prop key="administrator">administrator@example.com</prop>
                <prop key="support">support@example.com</prop>
            </props>
        </property>
    </bean>
    <bean id="child" parent="parent">
        <property name="adminEmails">
            <!-- the merge is specified on the child collection definition -->
            <props merge="true">
                <prop key="sales">sales@example.com</prop>
                <prop key="support">support@example.co.uk</prop>
            </props>
        </property>
    </bean>
<beans>
```

Notice the use of the `merge=true` attribute on the `<props/>` element of the `adminEmails` property of the `child` bean definition. When the `child` bean is resolved and instantiated by the container, the resulting instance has an `adminEmails` `Properties` collection that contains the result of merging the child's `adminEmails` collection with the parent's `adminEmails` collection. The following listing shows the result:

```
administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk
```

The child `Properties` collection's value set inherits all property elements from the parent `<props/>`, and the child's value for the `support` value overrides the value in the parent collection.

This merging behavior applies similarly to the `<list/>`, `<map/>`, and `<set/>` collection types. In the specific case of the `<list/>` element, the semantics associated with the `List` collection type (that is, the notion of an `ordered` collection of values) is maintained. The parent's values precede all of the child list's values. In the case of the `Map`, `Set`, and `Properties` collection types, no ordering exists. Hence, no ordering semantics are in effect for the collection types that underlie the associated `Map`, `Set`, and `Properties` implementation types that the container uses internally.

Limitations of Collection Merging

You cannot merge different collection types (such as a `Map` and a `List`). If you do attempt to do so, an appropriate `Exception` is thrown. The `merge` attribute must be specified on the lower, inherited, child definition. Specifying the `merge` attribute on a parent collection definition is redundant and does not result in the desired merging.

Strongly-typed collection

With the introduction of generic types in Java 5, you can use strongly typed collections. That is, it is possible to declare a `Collection` type such that it can only contain (for example) `String` elements. If you use Spring to dependency-inject a strongly-typed `Collection` into a bean, you can take advantage of Spring's type-conversion support such that the elements of your strongly-typed `Collection` instances are converted to the appropriate type prior to being added to the `Collection`. The following Java class and bean definition show how to do so:

```
public class SomeClass {

    private Map<String, Float> accounts;

    public void setAccounts(Map<String, Float> accounts) {
        this.accounts = accounts;
    }
}
```

```
<beans>
    <bean id="something" class="x.y.SomeClass">
        <property name="accounts">
            <map>
                <entry key="one" value="9.99"/>
                <entry key="two" value="2.75"/>
                <entry key="six" value="3.99"/>
            </map>
        </property>
    </bean>
</beans>
```

When the `accounts` property of the `something` bean is prepared for injection, the generics information about the element type of the strongly-typed `Map<String, Float>` is available by reflection. Thus, Spring's type conversion infrastructure recognizes the various value elements as being of type `Float`, and the string values (`9.99, 2.75`, and `3.99`) are converted into an actual `Float` type.

Null and Empty String Values

Spring treats empty arguments for properties and the like as empty `Strings`. The following XML-based configuration metadata snippet sets the `email` property to the empty `String` value ("").

```
<bean class="ExampleBean">
    <property name="email" value=""/>
</bean>
```

The preceding example is equivalent to the following Java code:

```
exampleBean.setEmail("");
```

The `<null/>` element handles `null` values. The following listing shows an example:

```
<bean class="ExampleBean">
    <property name="email">
        <null/>
    </property>
</bean>
```

The preceding configuration is equivalent to the following Java code:

```
exampleBean.setEmail(null);
```

XML Shortcut with the p-namespace

The p-namespace lets you use the `bean` element's attributes (instead of nested `<property/>` elements) to describe your property values collaborating beans, or both.

Spring supports extensible configuration formats with namespaces, which are based on an XML Schema definition. The `beans` configuration format discussed in this chapter is defined in an XML Schema document. However, the p-namespace is not defined in an XSD file and exists only in the core of Spring.

The following example shows two XML snippets (the first uses standard XML format and the second uses the p-namespace) that resolve to the same result:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="classic" class="com.example.ExampleBean">
        <property name="email" value="someone@somewhere.com"/>
    </bean>

    <bean name="p-namespace" class="com.example.ExampleBean"
        p:email="someone@somewhere.com"/>
</beans>
```

The example shows an attribute in the p-namespace called `email` in the bean definition. This tells Spring to include a property declaration. As previously mentioned, the p-namespace does not have a schema definition, so you can set the name of the attribute to the property name.

This next example includes two more bean definitions that both have a reference to another bean:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="john-classic" class="com.example.Person">
        <property name="name" value="John Doe"/>
        <property name="spouse" ref="jane"/>
    </bean>

    <bean name="john-modern"
        class="com.example.Person"
        p:name="John Doe"
        p:spouse-ref="jane"/>

    <bean name="jane" class="com.example.Person">
        <property name="name" value="Jane Doe"/>
    </bean>
</beans>
```

This example includes not only a property value using the p-namespace but also uses a special format to declare property references. Whereas the first bean definition uses `<property name="spouse" ref="jane"/>` to create a reference from bean `john` to bean `jane`, the second bean definition uses `p:spouse-ref="jane"` as an attribute to do the exact same thing. In this case, `spouse` is the property name, whereas the `-ref` part indicates that this is not a straight value but rather a reference to another bean.

NOTE The p-namespace is not as flexible as the standard XML format. For example, the format for declaring property references clashes with properties that end in `Ref`, whereas the standard XML format does not. We recommend that you choose your approach carefully and communicate this to your team members to avoid producing XML documents that use all three approaches at the same time.

XML Shortcut with the c-namespace

Similar to the XML Shortcut with the p-namespace, the c-namespace, introduced in Spring 3.1, allows inlined attributes for configuring the constructor arguments rather then nested `constructor-arg` elements.

The following example uses the `c:` namespace to do the same thing as the from Constructor-based Dependency Injection:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="beanTwo" class="x.y.ThingTwo"/>
    <bean id="beanThree" class="x.y.ThingThree"/>

    <!-- traditional declaration with optional argument names -->
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg name="thingTwo" ref="beanTwo"/>
        <constructor-arg name="thingThree" ref="beanThree"/>
        <constructor-arg name="email" value="something@somewhere.com"/>
    </bean>

    <!-- c-namespace declaration with argument names -->
    <bean id="beanOne" class="x.y.ThingOne" c:thingTwo-ref="beanTwo"
        c:thingThree-ref="beanThree" c:email="something@somewhere.com"/>

</beans>
```

The `c:` namespace uses the same conventions as the `p:` one (a trailing `-ref` for bean references) for setting the constructor arguments by their names. Similarly, it needs to be declared in the XML file even though it is not defined in an XSD schema (it exists inside the Spring core).

For the rare cases where the constructor argument names are not available (usually if the bytecode was compiled without debugging information), you can use fallback to the argument indexes, as follows:

```
<!-- c-namespace index declaration -->
<bean id="beanOne" class="x.y.ThingOne" c:_0-ref="beanTwo" c:_1-ref="beanThree"
    c:_2="something@somewhere.com"/>
```

NOTE Due to the XML grammar, the index notation requires the presence of the leading _, as XML attribute names cannot start with a number (even though some IDEs allow it). A corresponding index notation is also available for `<constructor-arg>` elements but not commonly used since the plain order of declaration is usually sufficient there.

In practice, the constructor resolution mechanism is quite efficient in matching arguments, so unless you really need to, we recommend using the name notation through-out your configuration.

Compound Property Names

You can use compound or nested property names when you set bean properties, as long as all components of the path except the final property name are not `null`. Consider the following bean definition:

```
<bean id="something" class="things.ThingOne">
    <property name="fred.bob.sammy" value="123" />
</bean>
```

The `something` bean has a `fred` property, which has a `bob` property, which has a `sammy` property, and that final `sammy` property is being set to a value of `123`. In order for this to work, the `fred` property of `something` and the `bob` property of `fred` must not be `null` after the bean is constructed. Otherwise, a `NullPointerException` is thrown.

Using `depends-on`

If a bean is a dependency of another bean, that usually means that one bean is set as a property of another. Typically you accomplish this with the `<ref/>` element in XML-based configuration metadata. However, sometimes dependencies between beans are less direct. An example is when a static initializer in a class needs to be triggered, such as for database driver registration. The `depends-on` attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized. The following example uses the `depends-on` attribute to express a dependency on a single bean:

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>
<bean id="manager" class="ManagerBean" />
```

To express a dependency on multiple beans, supply a list of bean names as the value of the depends-on attribute (commas, whitespace, and semicolons are valid delimiters):

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">
    <property name="manager" ref="manager" />
</bean>

<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

NOTE The depends-on attribute can specify both an initialization-time dependency and, in the case of singleton beans only, a corresponding destruction-time dependency. Dependent beans that define a depends-on relationship with a given bean are destroyed first, prior to the given bean itself being destroyed. Thus, depends-on can also control shutdown order.

Lazy-initialized Beans

By default, ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process. Generally, this pre-instantiation is desirable, because errors in the configuration or surrounding environment are discovered immediately, as opposed to hours or even days later. When this behavior is not desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as being lazy-initialized. A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

In XML, this behavior is controlled by the lazy-init attribute on the <bean/> element, as the following example shows:

```
<bean id="lazy" class="com.something.ExpensiveToCreateBean" lazy-init="true"/>
<bean name="not.lazy" class="com.something.AnotherBean"/>
```

When the preceding configuration is consumed by an ApplicationContext, the lazy bean is not eagerly pre-instantiated when the ApplicationContext starts, whereas the not.lazy bean is eagerly pre-instantiated.

However, when a lazy-initialized bean is a dependency of a singleton bean that is not lazy-initialized, the ApplicationContext creates the lazy-initialized bean at startup, because it must satisfy the singleton's dependencies. The lazy-initialized bean is injected into a singleton bean elsewhere that is not lazy-initialized.

You can also control lazy-initialization at the container level by using the `default-lazy-init` attribute on the `<beans/>` element, a the following example shows:

```
<beans default-lazy-init="true">
    <!-- no beans will be pre-instantiated... -->
</beans>
```

Autowiring Collaborators

The Spring container can autowire relationships between collaborating beans. You can let Spring resolve collaborators (other beans) automatically for your bean by inspecting the contents of the `ApplicationContext`. Autowiring has the following advantages:

- Autowiring can significantly reduce the need to specify properties or constructor arguments. (Other mechanisms such as a bean template discussed elsewhere in this chapter are also valuable in this regard.)
- Autowiring can update a configuration as your objects evolve. For example, if you need to add a dependency to a class, that dependency can be satisfied automatically without you needing to modify the configuration. Thus autowiring can be especially useful during development, without negating the option of switching to explicit wiring when the code base becomes more stable.

When using XML-based configuration metadata (see Dependency Injection), you can specify the autowire mode for a bean definition with the `autowire` attribute of the `<bean/>` element. The autowiring functionality has four modes. You specify autowiring per bean and can thus choose which ones to autowire. The following table describes the four autowiring modes:

Table 2. Autowiring modes

Mode	Explanation
`no`	(Default) No autowiring. Bean references must be defined by `ref` elements. Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity. To some extent, it documents the structure of a system.
`byName`	Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired. For example, if a bean definition is set to autowire by name and it contains a `master` property (that is, it has a `setMaster(..)` method), Spring looks for a bean definition named `master` and uses it to set the property.

Mode	Explanation
byType	Lets a property be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use byType autowiring for that bean. If there are no matching beans, nothing happens (the property is not set).
constructor	Analogous to byType but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

With byType or constructor autowiring mode, you can wire arrays and typed collections. In such cases, all autowire candidates within the container that match the expected type are provided to satisfy the dependency. You can autowire strongly-typed Map instances if the expected key type is String. An autowired Map instance's values consist of all bean instances that match the expected type, and the Map instance's keys contain the corresponding bean names.

Limitations and Disadvantages of Autowiring

Autowiring works best when it is used consistently across a project. If autowiring is not used in general, it might be confusing to developers to use it to wire only one or two bean definitions.

Consider the limitations and disadvantages of autowiring:

- Explicit dependencies in property and constructor-arg settings always override autowiring. You cannot autowire simple properties such as primitives, Strings, and Classes (and arrays of such simple properties). This limitation is by-design.
- Autowiring is less exact than explicit wiring. Although, as noted in the earlier table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results. The relationships between your Spring-managed objects are no longer documented explicitly.
- Wiring information may not be available to tools that may generate documentation from a Spring container.
- Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Map instances, this is not necessarily a problem. However, for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

In the latter scenario, you have several options:

- Abandon autowiring in favor of explicit wiring.
- Avoid autowiring for a bean definition by setting its autowire-candidate attributes to false, as described in the next section.
- Designate a single bean definition as the primary candidate by setting the primary attribute of its <bean/> element to true.
- Implement the more fine-grained control available with annotation-based configuration, as described in Annotation-based Container Configuration.

Excluding a Bean from Autowiring

On a per-bean basis, you can exclude a bean from autowiring. In Spring's XML format, set the `autowire-candidate` attribute of the `<bean/>` element to `false`. The container makes that specific bean definition unavailable to the autowiring infrastructure (including annotation style configurations such as `@Autowired`).

> **NOTE** The `autowire-candidate` attribute is designed to only affect type-based autowiring. It does not affect explicit references by name, which get resolved even if the specified bean is not marked as an autowire candidate. As a consequence, autowiring by name nevertheless injects a bean if the name matches.

You can also limit autowire candidates based on pattern-matching against bean names. The top-level `<beans/>` element accepts one or more patterns within its `default-autowire-candidates` attribute. For example, to limit autowire candidate status to any bean whose name ends with `Repository`, provide a value of `*Repository`. To provide multiple patterns, define them in a comma-separated list. An explicit value of `true` or `false` for a bean definition's `autowire-candidate` attribute always takes precedence. For such beans, the pattern matching rules do not apply.

These techniques are useful for beans that you never want to be injected into other beans by autowiring. It does not mean that an excluded bean cannot itself be configured by using autowiring. Rather, the bean itself is not a candidate for autowiring other beans.

Method Injection

In most application scenarios, most beans in the container are singletons. When a singleton bean needs to collaborate with another singleton bean or a non-singleton bean needs to collaborate with another non-singleton bean, you typically handle the dependency by defining one bean as a property of the other. A problem arises when the bean lifecycles are different. Suppose singleton bean A needs to use non-singleton (prototype) bean B, perhaps on each method invocation on A. The container creates the singleton bean A only once, and thus only gets one opportunity to set the properties. The container cannot provide bean A with a new instance of bean B every time one is needed.

A solution is to forego some inversion of control. You can make bean A aware of the container by implementing the `ApplicationContextAware` interface, and by making a `getBean("B")` call to the container ask for (a typically new) bean B instance every time bean A needs it. The following example shows this approach:

```java
// a class that uses a stateful Command-style class to perform some processing
package fiona.apple;

// Spring-API imports
import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;

public class CommandManager implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public Object process(Map commandState) {
        // grab a new instance of the appropriate Command
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    protected Command createCommand() {
        // notice the Spring API dependency!
        return this.applicationContext.getBean("command", Command.class);
    }

    public void setApplicationContext(
            ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

The preceding is not desirable, because the business code is aware of and coupled to the Spring Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, lets you handle this use case cleanly.

> You can read more about the motivation for Method Injection in this blog entry.

Lookup Method Injection

Lookup method injection is the ability of the container to override methods on container-managed beans and return the lookup result for another named bean in the container. The lookup typically involves a prototype bean, as in the scenario described in the preceding section. The Spring Framework implements this method injection by using bytecode generation from the CGLIB library to dynamically generate a subclass that overrides the method.

NOTE

- For this dynamic subclassing to work, the class that the Spring bean container subclasses cannot be `final`, and the method to be overridden cannot be `final`, either.
- Unit-testing a class that has an `abstract` method requires you to subclass the class yourself and to supply a stub implementation of the `abstract` method.
- Concrete methods are also necessary for component scanning, which requires concrete classes to pick up.
- A further key limitation is that lookup methods do not work with factory methods and in particular not with `@Bean` methods in configuration classes, since, in that case, the container is not in charge of creating the instance and therefore cannot create a runtime-generated subclass on the fly.

In the case of the `CommandManager` class in the previous code snippet, the Spring container dynamically overrides the implementation of the `createCommand()` method. The `CommandManager` class does not have any Spring dependencies, as the reworked example shows:

```
package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    // okay... but where is the implementation of this method?
    protected abstract Command createCommand();
}
```

In the client class that contains the method to be injected (the `CommandManager` in this case), the method to be injected requires a signature of the following form:

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

If the method is `abstract`, the dynamically-generated subclass implements the method. Otherwise, the dynamically-generated subclass overrides the concrete method defined in the original class. Consider the following example:

```xml
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="myCommand" class="fiona.apple.AsyncCommand" scope="prototype">
    <!-- inject dependencies here as required -->
</bean>

<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
    <lookup-method name="createCommand" bean="myCommand"/>
</bean>
```

The bean identified as `commandManager` calls its own `createCommand()` method whenever it needs a new instance of the `myCommand` bean. You must be careful to deploy the `myCommand` bean as a prototype if that is actually what is needed. If it is a singleton, the same instance of the `myCommand` bean is returned each time.

Alternatively, within the annotation-based component model, you can declare a lookup method through the `@Lookup` annotation, as the following example shows:

```java
public abstract class CommandManager {

    public Object process(Object commandState) {
        Command command = createCommand();
        command.setState(commandState);
        return command.execute();
    }

    @Lookup("myCommand")
    protected abstract Command createCommand();
}
```

Or, more idiomatically, you can rely on the target bean getting resolved against the declared return type of the lookup method:

```java
public abstract class CommandManager {

    public Object process(Object commandState) {
        MyCommand command = createCommand();
        command.setState(commandState);
        return command.execute();
    }

    @Lookup
    protected abstract MyCommand createCommand();
}
```

Note that you should typically declare such annotated lookup methods with a concrete stub implementation, in order for them to be compatible with Spring's component scanning rules where abstract classes get ignored by default. This limitation does not apply to explicitly registered or explicitly imported bean classes.

TIP

Another way of accessing differently scoped target beans is an `ObjectFactory`/ `Provider` injection point. See Scoped Beans as Dependencies.

You may also find the `ServiceLocatorFactoryBean` (in the `org.springframework.beans.factory.config` package) to be useful.

Arbitrary Method Replacement

A less useful form of method injection than lookup method injection is the ability to replace arbitrary methods in a managed bean with another method implementation. You can safely skip the rest of this section until you actually need this functionality.

With XML-based configuration metadata, you can use the `replaced-method` element to replace an existing method implementation with another, for a deployed bean. Consider the following class, which has a method called `computeValue` that we want to override:

```java
public class MyValueCalculator {

    public String computeValue(String input) {
        // some real code...
    }

    // some other methods...
}
```

A class that implements the `org.springframework.beans.factory.support.MethodReplacer` interface provides the new method definition, as the following example shows:

```java
/**
 * meant to be used to override the existing computeValue(String)
 * implementation in MyValueCalculator
 */
public class ReplacementComputeValue implements MethodReplacer {

    public Object reimplement(Object o, Method m, Object[] args) throws Throwable
{
        // get the input value, work with it, and return a computed result
        String input = (String) args[0];
        ...
        return ...;
    }
}
```

The bean definition to deploy the original class and specify the method override would resemble the following example:

```xml
<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">
    <!-- arbitrary method replacement -->
    <replaced-method name="computeValue" replacer="replacementComputeValue">
        <arg-type>String</arg-type>
    </replaced-method>
</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>
```

You can use one or more `<arg-type/>` elements within the `<replaced-method/>` element to indicate the method signature of the method being overridden. The signature for the arguments is necessary only if the method is overloaded and multiple variants exist within the class. For convenience, the type string for an argument may be a substring of the fully qualified type name. For example, the following all match `java.lang.String`:

```
java.lang.String
String
Str
```

Because the number of arguments is often enough to distinguish between each possible choice, this shortcut can save a lot of typing, by letting you type only the shortest string that matches an argument type.

Bean Scopes

When you create a bean definition, you create a recipe for creating actual instances of the class defined by that bean definition. The idea that a bean definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition but also control the scope of the objects created from a particular bean definition. This approach is powerful and flexible, because you can choose the scope of the objects you create through configuration instead of having to bake in the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes. The Spring Framework supports six scopes, four of which are available only if you use a web-aware `ApplicationContext`. You can also create a custom scope.

The following table describes the supported scopes:

Table 3. Bean scopes

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of an HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.
application	Scopes a single bean definition to the lifecycle of a `ServletContext`. Only valid in the context of a web-aware Spring `ApplicationContext`.
websocket	Scopes a single bean definition to the lifecycle of a `WebSocket`. Only valid in the context of a web-aware Spring `ApplicationContext`.

NOTE

As of Spring 3.0, a thread scope is available but is not registered by default: see {api-spring-framework}/context/support/SimpleThreadScope.html[`SimpleThreadScope`]. As of Spring 4.2, a transaction scope is available as well: {api-spring-framework}/transaction/support/SimpleTransactionScope.html[`SimpleTransactionScope`]. For instructions on how to register these or any other custom scopes, see Using a Custom Scope.

The Singleton Scope

Only one shared instance of a singleton bean is managed, and all requests for beans with an ID or IDs that match that bean definition result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object. The following image shows how the singleton scope works:

[singleton] | *images/singleton.png*

Spring's concept of a singleton bean differs from the singleton pattern as defined in the Gang of Four (GoF) patterns book. The GoF singleton hard-codes the scope of an object such that one and only one instance of a particular class is created per ClassLoader. The scope of the Spring singleton is best described as being per-container and per-bean. This means that, if you define one bean for a particular class in a single Spring container, the Spring container creates one and only one instance of the class defined by that bean definition. The singleton scope is the default scope in Spring. To define a bean as a singleton in XML, you can define a bean as shown in the following example:

```
<bean id="accountService" class="com.something.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the
default) -->
<bean id="accountService" class="com.something.DefaultAccountService" scope=
"singleton"/>
```

The Prototype Scope

The non-singleton prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made. That is, the bean is injected into another bean or you request it through a `getBean()` method call on the container. As a rule, you should use the prototype scope for all stateful beans and the singleton scope for stateless beans.

The following diagram illustrates the Spring prototype scope:

[prototype] | *images/prototype.png*

(A data access object (DAO) is not typically configured as a prototype, because a typical DAO does not hold any conversational state. It was easier for us to reuse the core of the singleton diagram.)

The following example defines a bean as a prototype in XML:

```
<bean id="accountService" class="com.something.DefaultAccountService" scope=
"prototype"/>
```

In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean. The container instantiates, configures, and otherwise assembles a prototype object and hands it to the client, with no further record of that prototype instance. Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, in the case of prototypes, configured destruction lifecycle callbacks are not called. The client code must clean up prototype-scoped objects and release expensive resources that the prototype beans hold. To get the Spring container to release resources held by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans that need to be cleaned up.

In some respects, the Spring container's role in regard to a prototype-scoped bean is a replacement for the Java `new` operator. All lifecycle management past that point must be handled by the client. (For details on the lifecycle of a bean in the Spring container, see Lifecycle Callbacks.)

Singleton Beans with Prototype-bean Dependencies

When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time. Thus, if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.

However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container instantiates the singleton bean and resolves and injects its dependencies. If you need a new instance of a prototype bean at runtime more than once, see Method Injection

Request, Session, Application, and WebSocket Scopes

The `request`, `session`, `application`, and `websocket` scopes are available only if you use a web-aware Spring `ApplicationContext` implementation (such as `XmlWebApplicationContext`). If you use these scopes with regular Spring IoC containers, such as the `ClassPathXmlApplicationContext`, an `IllegalStateException` that complains about an unknown bean scope is thrown.

Initial Web Configuration

To support the scoping of beans at the `request`, `session`, `application`, and `websocket` levels (web-scoped beans), some minor initial configuration is required before you define your beans. (This initial setup is not required for the standard scopes: `singleton` and `prototype`.)

How you accomplish this initial setup depends on your particular Servlet environment.

If you access scoped beans within Spring Web MVC, in effect, within a request that is processed by the Spring `DispatcherServlet`, no special setup is necessary. `DispatcherServlet` already exposes all relevant state.

If you use a Servlet 2.5 web container, with requests processed outside of Spring's `DispatcherServlet` (for example, when using JSF or Struts), you need to register the `org.springframework.web.context.request.RequestContextListener` `ServletRequestListener`. For Servlet 3.0+, this can be done programmatically by using the `WebApplicationInitializer` interface.

Alternatively, or for older containers, add the following declaration to your web application's `web.xml` file:

```
<web-app>
    ...
    <listener>
        <listener-class>
            org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>
    ...
</web-app>
```

Alternatively, if there are issues with your listener setup, consider using Spring's `RequestContextFilter`. The filter mapping depends on the surrounding web application configuration, so you have to change it as appropriate. The following listing shows the filter part of a web application:

```
<web-app>
    ...
    <filter>
        <filter-name>requestContextFilter</filter-name>
        <filter-class>org.springframework.web.filter.RequestContextFilter</filter-
class>
    </filter>
    <filter-mapping>
        <filter-name>requestContextFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

`DispatcherServlet`, `RequestContextListener`, and `RequestContextFilter` all do exactly the same thing, namely bind the HTTP request object to the `Thread` that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain.

Request scope

Consider the following XML configuration for a bean definition:

```
<bean id="loginAction" class="com.something.LoginAction" scope="request"/>
```

The Spring container creates a new instance of the `LoginAction` bean by using the `loginAction` bean definition for each and every HTTP request. That is, the `loginAction` bean is scoped at the HTTP request level. You can change the internal state of the instance that is created as much as you want, because other instances created from the same `loginAction` bean definition do not see these changes in state. They are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.

When using annotation-driven components or Java configuration, the `@RequestScope` annotation can be used to assign a component to the `request` scope. The following example shows how to do so:

```
@RequestScope
@Component
public class LoginAction {
    // ...
}
```

Session Scope

Consider the following XML configuration for a bean definition:

```
<bean id="userPreferences" class="com.something.UserPreferences" scope="session"/>
```

The Spring container creates a new instance of the `UserPreferences` bean by using the `userPreferences` bean definition for the lifetime of a single HTTP `Session`. In other words, the `userPreferences` bean is effectively scoped at the HTTP `Session` level. As with request-scoped beans, you can change the internal state of the instance that is created as much as you want, knowing that other HTTP `Session` instances that are also using instances created from the same `userPreferences` bean definition do not see these changes in state, because they are particular to an individual HTTP `Session`. When the HTTP `Session` is eventually discarded, the bean that is scoped to that particular HTTP `Session` is also discarded.

When using annotation-driven components or Java configuration, you can use the `@SessionScope` annotation to assign a component to the `session` scope.

```
@SessionScope
@Component
public class UserPreferences {
    // ...
}
```

Application Scope

Consider the following XML configuration for a bean definition:

```
<bean id="appPreferences" class="com.something.AppPreferences" scope="application
"/>
```

The Spring container creates a new instance of the `AppPreferences` bean by using the `appPreferences` bean definition once for the entire web application. That is, the `appPreferences` bean is scoped at the `ServletContext` level and stored as a regular `ServletContext` attribute. This is somewhat similar to a Spring singleton bean but differs in two important ways: It is a singleton per `ServletContext`, not per Spring 'ApplicationContext' (for which there may be several in any given web application), and it is actually exposed and therefore visible as a `ServletContext` attribute.

When using annotation-driven components or Java configuration, you can use the `@ApplicationScope` annotation to assign a component to the `application` scope. The following example shows how to do so:

```
@ApplicationScope
@Component
public class AppPreferences {
    // ...
}
```

Scoped Beans as Dependencies

The Spring IoC container manages not only the instantiation of your objects (beans), but also the wiring up of collaborators (or dependencies). If you want to inject (for example) an HTTP request-scoped bean into another bean of a longer-lived scope, you may choose to inject an AOP proxy in place of the scoped bean. That is, you need to inject a proxy object that exposes the same public interface as the scoped object but that can also retrieve the real target object from the relevant scope (such as an HTTP request) and delegate method calls onto the real object.

NOTE

You may also use `<aop:scoped-proxy/>` between beans that are scoped as `singleton`, with the reference then going through an intermediate proxy that is serializable and therefore able to re-obtain the target singleton bean on deserialization.

When declaring `<aop:scoped-proxy/>` against a bean of scope `prototype`, every method call on the shared proxy leads to the creation of a new target instance to which the call is then being forwarded.

Also, scoped proxies are not the only way to access beans from shorter scopes in a lifecycle-safe fashion. You may also declare your injection point (that is, the constructor or setter argument or autowired field) as `ObjectFactory<MyTargetBean>`, allowing for a `getObject()` call to retrieve the current instance on demand every time it is needed — without holding on to the instance or storing it separately.

As an extended variant, you may declare `ObjectProvider<MyTargetBean>`, which delivers several additional access variants, including `getIfAvailable` and `getIfUnique`.

The JSR-330 variant of this is called `Provider` and is used with a `Provider<MyTargetBean>` declaration and a corresponding `get()` call for every retrieval attempt. See here for more details on JSR-330 overall.

The configuration in the following example is only one line, but it is important to understand the “why” as well as the “how” behind it:

```
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!-- an HTTP Session-scoped bean exposed as a proxy -->
    <bean id="userPreferences" class="com.something.UserPreferences" scope=
"session">
        <!-- instructs the container to proxy the surrounding bean -->
        <aop:scoped-proxy/> ①
    </bean>

    <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userService" class="com.something.SimpleUserService">
        <!-- a reference to the proxied userPreferences bean -->
        <property name="userPreferences" ref="userPreferences"/>
    </bean>
</beans>
```

① The line that defines the proxy.

To create such a proxy, you insert a child `<aop:scoped-proxy/>` element into a scoped bean definition (see Choosing the Type of Proxy to Create and XML Schema-based configuration). Why do definitions of beans scoped at the `request`, `session` and custom-scope levels require the `<aop:scoped-proxy/>` element? Consider the following singleton bean definition and contrast it with what you need to define for the aforementioned scopes (note that the following `userPreferences` bean definition as it stands is incomplete):

```
<bean id="userPreferences" class="com.something.UserPreferences" scope="session"/>

<bean id="userManager" class="com.something.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

In the preceding example, the singleton bean (`userManager`) is injected with a reference to the HTTP `Session`-scoped bean (`userPreferences`). The salient point here is that the `userManager` bean is a singleton: it is instantiated exactly once per container, and its dependencies (in this case only one, the `userPreferences` bean) are also injected only once. This means that the `userManager` bean operates only on the exact same `userPreferences` object (that is, the one with which it was originally injected.

This is not the behavior you want when injecting a shorter-lived scoped bean into a longer-lived scoped bean (for example, injecting an HTTP `Session`-scoped collaborating bean as a dependency into singleton bean). Rather, you need a single `userManager` object, and, for the lifetime of an HTTP `Session`, you need a `userPreferences` object that is specific to the HTTP `Session`. Thus, the container creates an object that exposes the exact same public interface as the `UserPreferences` class (ideally an object that is a `UserPreferences` instance), which can fetch the real `UserPreferences` object from the scoping mechanism (HTTP request, `Session`, and so forth). The container injects this proxy object into the `userManager` bean, which is unaware that this `UserPreferences` reference is a proxy. In this example, when a `UserManager` instance invokes a method on the dependency-injected `UserPreferences` object, it is actually invoking a method on the proxy. The proxy then fetches the real `UserPreferences` object from (in this case) the HTTP `Session` and delegates the method invocation onto the retrieved real `UserPreferences` object.

Thus, you need the following (correct and complete) configuration when injecting `request-` and `session-scoped` beans into collaborating objects, as the following example shows:

```
<bean id="userPreferences" class="com.something.UserPreferences" scope="session">
    <aop:scoped-proxy/>
</bean>

<bean id="userManager" class="com.something.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

Choosing the Type of Proxy to Create

By default, when the Spring container creates a proxy for a bean that is marked up with the `<aop:scoped-proxy/>` element, a CGLIB-based class proxy is created.

NOTE	CGLIB proxies intercept only public method calls! Do not call non-public methods on such a proxy. They are not delegated to the actual scoped target object.

Alternatively, you can configure the Spring container to create standard JDK interface-based proxies for such scoped beans, by specifying `false` for the value of the `proxy-target-class` attribute of the `<aop:scoped-proxy/>` element. Using JDK interface-based proxies means that you do not need additional libraries in your application classpath to affect such proxying. However, it also means that the class of the scoped bean must implement at least one interface and that all collaborators into which the scoped bean is injected must reference the bean through one of its interfaces. The following example shows a proxy based on an interface:

```xml
<!-- DefaultUserPreferences implements the UserPreferences interface -->
<bean id="userPreferences" class="com.stuff.DefaultUserPreferences" scope="
session">
    <aop:scoped-proxy proxy-target-class="false"/>
</bean>

<bean id="userManager" class="com.stuff.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

For more detailed information about choosing class-based or interface-based proxying, see [aop-proxying].

Custom Scopes

The bean scoping mechanism is extensible. You can define your own scopes or even redefine existing scopes, although the latter is considered bad practice and you cannot override the built-in `singleton` and `prototype` scopes.

Creating a Custom Scope

To integrate your custom scopes into the Spring container, you need to implement the `org.springframework.beans.factory.config.Scope` interface, which is described in this section. For an idea of how to implement your own scopes, see the `Scope` implementations that are supplied with the Spring Framework itself and the {api-spring-framework}/beans/factory/config/Scope.html[`Scope`] javadoc, which explains the methods you need to implement in more detail.

The `Scope` interface has four methods to get objects from the scope, remove them from the scope, and let them be destroyed.

The session scope implementation, for example, returns the session-scoped bean (if it does not exist, the method returns a new instance of the bean, after having bound it to the session for future reference). The following method returns the object from the underlying scope:

```java
Object get(String name, ObjectFactory objectFactory)
```

The session scope implementation, for example, removes the session-scoped bean from the underlying session. The object should be returned, but you can return null if the object with the specified name is not found. The following method removes the object from the underlying scope:

```java
Object remove(String name)
```

The following method registers the callbacks the scope should execute when it is destroyed or when the specified object in the scope is destroyed:

```
void registerDestructionCallback(String name, Runnable destructionCallback)
```

See the {api-spring-framework}/beans/factory/config/Scope.html#registerDestructionCallback[javadoc] or a Spring scope implementation for more information on destruction callbacks.

The following method obtains the conversation identifier for the underlying scope:

```
String getConversationId()
```

This identifier is different for each scope. For a session scoped implementation, this identifier can be the session identifier.

Using a Custom Scope

After you write and test one or more custom `Scope` implementations, you need to make the Spring container aware of your new scopes. The following method is the central method to register a new `Scope` with the Spring container:

```
void registerScope(String scopeName, Scope scope);
```

This method is declared on the `ConfigurableBeanFactory` interface, which is available through the `BeanFactory` property on most of the concrete `ApplicationContext` implementations that ship with Spring.

The first argument to the `registerScope(..)` method is the unique name associated with a scope. Examples of such names in the Spring container itself are `singleton` and `prototype`. The second argument to the `registerScope(..)` method is an actual instance of the custom `Scope` implementation that you wish to register and use.

Suppose that you write your custom `Scope` implementation, and then register it as shown in the next example.

NOTE	The next example uses `SimpleThreadScope`, which is included with Spring but is not registered by default. The instructions would be the same for your own custom `Scope` implementations.

```java
Scope threadScope = new SimpleThreadScope();
beanFactory.registerScope("thread", threadScope);
```

You can then create bean definitions that adhere to the scoping rules of your custom `Scope`, as follows:

```xml
<bean id="..." class="..." scope="thread">
```

With a custom `Scope` implementation, you are not limited to programmatic registration of the scope. You can also do the `Scope` registration declaratively, by using the `CustomScopeConfigurer` class, as the following example shows:

```xml
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
        https://www.springframework.org/schema/aop/spring-aop.xsd">

    <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="thread">
                    <bean class=
"org.springframework.context.support.SimpleThreadScope"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="thing2" class="x.y.Thing2" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>

    <bean id="thing1" class="x.y.Thing1">
        <property name="thing2" ref="thing2"/>
    </bean>

</beans>
```

NOTE When you place `<aop:scoped-proxy/>` in a `FactoryBean` implementation, it is the factory bean itself that is scoped, not the object returned from `getObject()`.

Customizing the Nature of a Bean

The Spring Framework provides a number of interfaces you can use to customize the nature of a bean. This section groups them as follows:

- Lifecycle Callbacks
- `ApplicationContextAware` and `BeanNameAware`
- Other `Aware` Interfaces

Lifecycle Callbacks

To interact with the container's management of the bean lifecycle, you can implement the Spring `InitializingBean` and `DisposableBean` interfaces. The container calls `afterPropertiesSet()` for the former and `destroy()` for the latter to let the bean perform certain actions upon initialization and destruction of your beans.

TIP The JSR-250 `@PostConstruct` and `@PreDestroy` annotations are generally considered best practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations means that your beans are not coupled to Spring-specific interfaces. For details, see Using `@PostConstruct` and `@PreDestroy`.

If you do not want to use the JSR-250 annotations but you still want to remove coupling, consider `init-method` and `destroy-method` bean definition metadata.

Internally, the Spring Framework uses `BeanPostProcessor` implementations to process any callback interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring does not by default offer, you can implement a `BeanPostProcessor` yourself. For more information, see Container Extension Points.

In addition to the initialization and destruction callbacks, Spring-managed objects may also implement the `Lifecycle` interface so that those objects can participate in the startup and shutdown process, as driven by the container's own lifecycle.

The lifecycle callback interfaces are described in this section.

Initialization Callbacks

The `org.springframework.beans.factory.InitializingBean` interface lets a bean perform initialization work after the container has set all necessary properties on the bean. The `InitializingBean` interface specifies a single method:

```java
void afterPropertiesSet() throws Exception;
```

We recommend that you do not use the `InitializingBean` interface, because it unnecessarily couples the code to Spring. Alternatively, we suggest using the `@PostConstruct` annotation or specifying a POJO initialization method. In the case of XML-based configuration metadata, you can use the `init-method` attribute to specify the name of the method that has a void no-argument signature. With Java configuration, you can use the `initMethod` attribute of `@Bean`. See Receiving Lifecycle Callbacks. Consider the following example:

```xml
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

```java
public class ExampleBean {

    public void init() {
        // do some initialization work
    }
}
```

The preceding example has almost exactly the same effect as the following example (which consists of two listings):

```xml
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```java
public class AnotherExampleBean implements InitializingBean {

    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

However, the first of the two preceding examples does not couple the code to Spring.

Destruction Callbacks

Implementing the `org.springframework.beans.factory.DisposableBean` interface lets a bean get a callback when the container that contains it is destroyed. The `DisposableBean` interface specifies a single method:

```java
void destroy() throws Exception;
```

We recommend that you do not use the `DisposableBean` callback interface, because it unnecessarily couples the code to Spring. Alternatively, we suggest using the `@PreDestroy` annotation or specifying a generic method that is supported by bean definitions. With XML-based configuration metadata, you can use the `destroy-method` attribute on the `<bean/>`. With Java configuration, you can use the `destroyMethod` attribute of `@Bean`. See Receiving Lifecycle Callbacks. Consider the following definition:

```xml
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

```java
public class ExampleBean {

    public void cleanup() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

The preceding definition has almost exactly the same effect as the following definition:

```xml
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```java
public class AnotherExampleBean implements DisposableBean {

    public void destroy() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

However, the first of the two preceding definitions does not couple the code to Spring.

TIP You can assign the `destroy-method` attribute of a `<bean>` element a special `(inferred)` value, which instructs Spring to automatically detect a public `close` or `shutdown` method on the specific bean class. (Any class that implements `java.lang.AutoCloseable` or `java.io.Closeable` would therefore match.) You can also set this special `(inferred)` value on the `default-destroy-method` attribute of a `<beans>` element to apply this behavior to an entire set of beans (see Default Initialization and Destroy Methods). Note that this is the default behavior with Java configuration.

Default Initialization and Destroy Methods

When you write initialization and destroy method callbacks that do not use the Spring-specific `InitializingBean` and `DisposableBean` callback interfaces, you typically write methods with names such as `init()`, `initialize()`, `dispose()`, and so on. Ideally, the names of such lifecycle callback methods are standardized across a project so that all developers use the same method names and ensure consistency.

You can configure the Spring container to "look" for named initialization and destroy callback method names on every bean. This means that you, as an application developer, can write your application classes and use an initialization callback called `init()`, without having to configure an `init-method="init"` attribute with each bean definition. The Spring IoC container calls that method when the bean is created (and in accordance with the standard lifecycle callback contract described previously). This feature also enforces a consistent naming convention for initialization and destroy method callbacks.

Suppose that your initialization callback methods are named `init()` and your destroy callback methods are named `destroy()`. Your class then resembles the class in the following example:

```
public class DefaultBlogService implements BlogService {

    private BlogDao blogDao;

    public void setBlogDao(BlogDao blogDao) {
        this.blogDao = blogDao;
    }

    // this is (unsurprisingly) the initialization callback method
    public void init() {
        if (this.blogDao == null) {
            throw new IllegalStateException("The [blogDao] property must be set."
);
        }
    }
}
```

You could then use that class in a bean resembling the following:

```
<beans default-init-method="init">

    <bean id="blogService" class="com.something.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>

</beans>
```

The presence of the `default-init-method` attribute on the top-level `<beans/>` element attribute causes the Spring IoC container to recognize a method called `init` on the bean class as the initialization method callback. When a bean is created and assembled, if the bean class has such a method, it is invoked at the appropriate time.

You can configure destroy method callbacks similarly (in XML, that is) by using the `default-destroy-method` attribute on the top-level `<beans/>` element.

Where existing bean classes already have callback methods that are named at variance with the convention, you can override the default by specifying (in XML, that is) the method name by using the `init-method` and `destroy-method` attributes of the `<bean/>` itself.

The Spring container guarantees that a configured initialization callback is called immediately after a bean is supplied with all dependencies. Thus, the initialization callback is called on the raw bean reference, which means that AOP interceptors and so forth are not yet applied to the bean. A target bean is fully created first and then an AOP proxy (for example) with its interceptor chain is applied. If the target bean and the proxy are defined separately, your code can even interact with the raw target bean, bypassing the proxy. Hence, it would be inconsistent to apply the interceptors to the `init` method, because doing so would couple the lifecycle of the target bean to its proxy or interceptors and leave strange semantics when your code interacts directly with the raw target bean.

Combining Lifecycle Mechanisms

As of Spring 2.5, you have three options for controlling bean lifecycle behavior:

- The `InitializingBean` and `DisposableBean` callback interfaces
- Custom `init()` and `destroy()` methods
- The `@PostConstruct` and `@PreDestroy` annotations. You can combine these mechanisms to control a given bean.

NOTE	If multiple lifecycle mechanisms are configured for a bean and each mechanism is configured with a different method name, then each configured method is executed in the order listed after this note. However, if the same method name is configured — for example, `init()` for an initialization method — for more than one of these lifecycle mechanisms, that method is executed once, as explained in the preceding section.

Multiple lifecycle mechanisms configured for the same bean, with different initialization methods, are called as follows:

1. Methods annotated with `@PostConstruct`
2. `afterPropertiesSet()` as defined by the `InitializingBean` callback interface
3. A custom configured `init()` method

Destroy methods are called in the same order:

1. Methods annotated with `@PreDestroy`

2. `destroy()` as defined by the `DisposableBean` callback interface
3. A custom configured `destroy()` method

Startup and Shutdown Callbacks

The `Lifecycle` interface defines the essential methods for any object that has its own lifecycle requirements (such as starting and stopping some background process):

```
public interface Lifecycle {

    void start();

    void stop();

    boolean isRunning();
}
```

Any Spring-managed object may implement the `Lifecycle` interface. Then, when the `ApplicationContext` itself receives start and stop signals (for example, for a stop/restart scenario at runtime), it cascades those calls to all `Lifecycle` implementations defined within that context. It does this by delegating to a `LifecycleProcessor`, shown in the following listing:

```
public interface LifecycleProcessor extends Lifecycle {

    void onRefresh();

    void onClose();
}
```

Notice that the `LifecycleProcessor` is itself an extension of the `Lifecycle` interface. It also adds two other methods for reacting to the context being refreshed and closed.

TIP

Note that the regular `org.springframework.context.Lifecycle` interface is a plain contract for explicit start and stop notifications and does not imply auto-startup at context refresh time. For fine-grained control over auto-startup of a specific bean (including startup phases), consider implementing `org.springframework.context.SmartLifecycle` instead.

Also, please note that stop notifications are not guaranteed to come before destruction. On regular shutdown, all `Lifecycle` beans first receive a stop notification before the general destruction callbacks are being propagated. However, on hot refresh during a context's lifetime or on aborted refresh attempts, only destroy methods are called.

The order of startup and shutdown invocations can be important. If a “depends-on” relationship exists between any two objects, the dependent side starts after its dependency, and it stops before its dependency. However, at times, the direct dependencies are unknown. You may only know that objects of a certain type should start prior to objects of another type. In those cases, the `SmartLifecycle` interface defines another option, namely the `getPhase()` method as defined on its super-interface, `Phased`. The following listing shows the definition of the `Phased` interface:

```java
public interface Phased {

    int getPhase();
}
```

The following listing shows the definition of the `SmartLifecycle` interface:

```java
public interface SmartLifecycle extends Lifecycle, Phased {

    boolean isAutoStartup();

    void stop(Runnable callback);
}
```

When starting, the objects with the lowest phase start first. When stopping, the reverse order is followed. Therefore, an object that implements `SmartLifecycle` and whose `getPhase()` method returns `Integer.MIN_VALUE` would be among the first to start and the last to stop. At the other end of the spectrum, a phase value of `Integer.MAX_VALUE` would indicate that the object should be started last and stopped first (likely because it depends on other processes to be running). When considering the phase value, it is also important to know that the default phase for any “normal” `Lifecycle` object that does not implement `SmartLifecycle` is `0`. Therefore, any negative phase value indicates that an object should start before those standard components (and stop after them). The reverse is true for any positive phase value.

The stop method defined by `SmartLifecycle` accepts a callback. Any implementation must invoke that callback’s `run()` method after that implementation’s shutdown process is complete. That enables asynchronous shutdown where necessary, since the default implementation of the `LifecycleProcessor` interface, `DefaultLifecycleProcessor`, waits up to its timeout value for the group of objects within each phase to invoke that callback. The default per-phase timeout is 30 seconds. You can override the default lifecycle processor instance by defining a bean named `lifecycleProcessor` within the context. If you want only to modify the timeout, defining the following would suffice:

```xml
<bean id="lifecycleProcessor" class=
"org.springframework.context.support.DefaultLifecycleProcessor">
    <!-- timeout value in milliseconds -->
    <property name="timeoutPerShutdownPhase" value="10000"/>
</bean>
```

As mentioned earlier, the `LifecycleProcessor` interface defines callback methods for the refreshing and closing of the context as well. The latter drives the shutdown process as if `stop()` had been called explicitly, but it happens when the context is closing. The 'refresh' callback, on the other hand, enables another feature of `SmartLifecycle` beans. When the context is refreshed (after all objects have been instantiated and initialized), that callback is invoked. At that point, the default lifecycle processor checks the boolean value returned by each `SmartLifecycle` object's `isAutoStartup()` method. If `true`, that object is started at that point rather than waiting for an explicit invocation of the context's or its own `start()` method (unlike the context refresh, the context start does not happen automatically for a standard context implementation). The `phase` value and any "depends-on" relationships determine the startup order as described earlier.

Shutting Down the Spring IoC Container Gracefully in Non-Web Applications

NOTE This section applies only to non-web applications. Spring's web-based `ApplicationContext` implementations already have code in place to gracefully shut down the Spring IoC container when the relevant web application is shut down.

If you use Spring's IoC container in a non-web application environment (for example, in a rich client desktop environment), register a shutdown hook with the JVM. Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released. You must still configure and implement these destroy callbacks correctly.

To register a shutdown hook, call the `registerShutdownHook()` method that is declared on the `ConfigurableApplicationContext` interface, as the following example shows:

```java
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        ConfigurableApplicationContext ctx = new ClassPathXmlApplicationContext(
"beans.xml");

        // add a shutdown hook for the above context...
        ctx.registerShutdownHook();

        // app runs here...

        // main method exits, hook is called prior to the app shutting down...
    }
}
```

`ApplicationContextAware` and `BeanNameAware`

When an `ApplicationContext` creates an object instance that implements the `org.springframework.context.ApplicationContextAware` interface, the instance is provided with a reference to that `ApplicationContext`. The following listing shows the definition of the `ApplicationContextAware` interface:

```java
public interface ApplicationContextAware {

    void setApplicationContext(ApplicationContext applicationContext) throws
BeansException;
}
```

Thus, beans can programmatically manipulate the `ApplicationContext` that created them, through the `ApplicationContext` interface or by casting the reference to a known subclass of this interface (such as `ConfigurableApplicationContext`, which exposes additional functionality). One use would be the programmatic retrieval of other beans. Sometimes this capability is useful. However, in general, you should avoid it, because it couples the code to Spring and does not follow the Inversion of Control style, where collaborators are provided to beans as properties. Other methods of the `ApplicationContext` provide access to file resources, publishing application events, and accessing a `MessageSource`. These additional features are described in Additional Capabilities of the `ApplicationContext`.

As of Spring 2.5, autowiring is another alternative to obtain a reference to the `ApplicationContext`. The "traditional" `constructor` and `byType` autowiring modes (as described in Autowiring Collaborators) can provide a dependency of type `ApplicationContext` for a constructor argument or a setter method parameter, respectively. For more flexibility, including the ability to autowire fields and multiple parameter methods, use the new annotation-based autowiring features. If you do, the `ApplicationContext` is autowired into a field, constructor argument, or method parameter that expects the `ApplicationContext` type if the field, constructor, or method in question carries the `@Autowired` annotation. For more information, see Using `@Autowired`.

When an `ApplicationContext` creates a class that implements the `org.springframework.beans.factory.BeanNameAware` interface, the class is provided with a reference to the name defined in its associated object definition. The following listing shows the definition of the BeanNameAware interface:

```
public interface BeanNameAware {

    void setBeanName(String name) throws BeansException;
}
```

The callback is invoked after population of normal bean properties but before an initialization callback such as `InitializingBean`, `afterPropertiesSet`, or a custom init-method.

Other `Aware` Interfaces

Besides `ApplicationContextAware` and `BeanNameAware` (discussed earlier), Spring offers a wide range of `Aware` callback interfaces that let beans indicate to the container that they require a certain infrastructure dependency. As a general rule, the name indicates the dependency type. The following table summarizes the most important `Aware` interfaces:

Table 4. Aware interfaces

Name	Injected Dependency	Explained in...
`ApplicationContextAware`	Declaring `ApplicationContext`.	`ApplicationContextAware` and `BeanNameAware`
`ApplicationEventPublisherAware`	Event publisher of the enclosing `ApplicationContext`.	Additional Capabilities of the `ApplicationContext`
`BeanClassLoaderAware`	Class loader used to load the bean classes.	Instantiating Beans
`BeanFactoryAware`	Declaring `BeanFactory`.	`ApplicationContextAware` and `BeanNameAware`
`BeanNameAware`	Name of the declaring bean.	`ApplicationContextAware` and `BeanNameAware`
`BootstrapContextAware`	Resource adapter `BootstrapContext` the container runs in. Typically available only in JCA-aware `ApplicationContext` instances.	JCA CCI

Name	Injected Dependency	Explained in…
`LoadTimeWeaverAware`	Defined weaver for processing class definition at load time.	[aop-aj-ltw]
`MessageSourceAware`	Configured strategy for resolving messages (with support for parametrization and internationalization).	Additional Capabilities of the `ApplicationContext`
`NotificationPublisherAware`	Spring JMX notification publisher.	Notifications
`ResourceLoaderAware`	Configured loader for low-level access to resources.	[resources]
`ServletConfigAware`	Current `ServletConfig` the container runs in. Valid only in a web-aware Spring `ApplicationContext`.	Spring MVC
`ServletContextAware`	Current `ServletContext` the container runs in. Valid only in a web-aware Spring `ApplicationContext`.	Spring MVC

Note again that using these interfaces ties your code to the Spring API and does not follow the Inversion of Control style. As a result, we recommend them for infrastructure beans that require programmatic access to the container.

Bean Definition Inheritance

A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information, such as the initialization method, a static factory method name, and so on. A child bean definition inherits configuration data from a parent definition. The child definition can override some values or add others as needed. Using parent and child bean definitions can save a lot of typing. Effectively, this is a form of templating.

If you work with an `ApplicationContext` interface programmatically, child bean definitions are represented by the `ChildBeanDefinition` class. Most users do not work with them on this level. Instead, they configure bean definitions declaratively in a class such as the `ClassPathXmlApplicationContext`. When you use XML-based configuration metadata, you can indicate a child bean definition by using the `parent` attribute, specifying the parent bean as the value of this attribute. The following example shows how to do so:

```xml
<bean id="inheritedTestBean" abstract="true"
        class="org.springframework.beans.TestBean">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
        class="org.springframework.beans.DerivedTestBean"
        parent="inheritedTestBean" init-method="initialize">  ①
    <property name="name" value="override"/>
    <!-- the age property value of 1 will be inherited from parent -->
</bean>
```

① Note the `parent` attribute.

A child bean definition uses the bean class from the parent definition if none is specified but can also override it. In the latter case, the child bean class must be compatible with the parent (that is, it must accept the parent's property values).

A child bean definition inherits scope, constructor argument values, property values, and method overrides from the parent, with the option to add new values. Any scope, initialization method, destroy method, or `static` factory method settings that you specify override the corresponding parent settings.

The remaining settings are always taken from the child definition: depends on, autowire mode, dependency check, singleton, and lazy init.

The preceding example explicitly marks the parent bean definition as abstract by using the `abstract` attribute. If the parent definition does not specify a class, explicitly marking the parent bean definition as `abstract` is required, as the following example shows:

```xml
<bean id="inheritedTestBeanWithoutClass" abstract="true">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"
        parent="inheritedTestBeanWithoutClass" init-method="initialize">
    <property name="name" value="override"/>
    <!-- age will inherit the value of 1 from the parent bean definition-->
</bean>
```

The parent bean cannot be instantiated on its own because it is incomplete, and it is also explicitly marked as `abstract`. When a definition is `abstract`, it is usable only as a pure template bean definition that serves as a parent definition for child definitions. Trying to use such an `abstract` parent bean on its own, by referring to it as a ref property of another bean or doing an explicit `getBean()` call with the parent bean ID returns an error. Similarly, the container's internal `preInstantiateSingletons()` method ignores bean definitions that are defined as abstract.

> **NOTE**
>
> `ApplicationContext` pre-instantiates all singletons by default. Therefore, it is important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the *abstract* attribute to *true*, otherwise the application context will actually (attempt to) pre-instantiate the `abstract` bean.

Container Extension Points

Typically, an application developer does not need to subclass `ApplicationContext` implementation classes. Instead, the Spring IoC container can be extended by plugging in implementations of special integration interfaces. The next few sections describe these integration interfaces.

Customizing Beans by Using a `BeanPostProcessor`

The `BeanPostProcessor` interface defines callback methods that you can implement to provide your own (or override the container's default) instantiation logic, dependency resolution logic, and so forth. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more custom `BeanPostProcessor` implementations.

You can configure multiple `BeanPostProcessor` instances, and you can control the order in which these `BeanPostProcessor` instances execute by setting the `order` property. You can set this property only if the `BeanPostProcessor` implements the `Ordered` interface. If you write your own `BeanPostProcessor`, you should consider implementing the `Ordered` interface, too. For further details, see the javadoc of the {api-spring-framework}/beans/factory/config/BeanPostProcessor.html[`BeanPostProcessor`] and {api-spring-framework}/core/Ordered.html[`Ordered`] interfaces. See also the note on programmatic registration of `BeanPostProcessor` instances.

> **NOTE**
>
> `BeanPostProcessor` instances operate on bean (or object) instances. That is, the Spring IoC container instantiates a bean instance and then `BeanPostProcessor` instances do their work.
>
> `BeanPostProcessor` instances are scoped per-container. This is relevant only if you use container hierarchies. If you define a `BeanPostProcessor` in one container, it post-processes only the beans in that container. In other words, beans that are defined in one container are not post-processed by a `BeanPostProcessor` defined in another container, even if both containers are part of the same hierarchy.
>
> To change the actual bean definition (that is, the blueprint that defines the bean), you instead need to use a `BeanFactoryPostProcessor`, as described in Customizing Configuration Metadata with a `BeanFactoryPostProcessor`.

The `org.springframework.beans.factory.config.BeanPostProcessor` interface consists of exactly two callback methods. When such a class is registered as a post-processor with the container, for each bean instance that is created by the container, the post-processor gets a callback from the container both before container initialization methods (such as `InitializingBean.afterPropertiesSet()` or any declared `init` method) are called, and after any bean initialization callbacks. The post-processor can take any action with the bean instance, including ignoring the callback completely. A bean post-processor typically checks for callback interfaces, or it may wrap a bean with a proxy. Some Spring AOP infrastructure classes are implemented as bean post-processors in order to provide proxy-wrapping logic.

An `ApplicationContext` automatically detects any beans that are defined in the configuration metadata that implements the `BeanPostProcessor` interface. The `ApplicationContext` registers these beans as post-processors so that they can be called later, upon bean creation. Bean post-processors can be deployed in the container in the same fashion as any other beans.

Note that, when declaring a `BeanPostProcessor` by using an `@Bean` factory method on a configuration class, the return type of the factory method should be the implementation class itself or at least the `org.springframework.beans.factory.config.BeanPostProcessor` interface, clearly indicating the post-processor nature of that bean. Otherwise, the `ApplicationContext` cannot autodetect it by type before fully creating it. Since a `BeanPostProcessor` needs to be instantiated early in order to apply to the initialization of other beans in the context, this early type detection is critical.

NOTE

Programmatically registering `BeanPostProcessor` *instances*

While the recommended approach for `BeanPostProcessor` registration is through `ApplicationContext` auto-detection (as described earlier), you can register them programmatically against a `ConfigurableBeanFactory` by using the `addBeanPostProcessor` method. This can be useful when you need to evaluate conditional logic before registration or even for copying bean post processors across contexts in a hierarchy. Note, however, that `BeanPostProcessor` instances added programmatically do not respect the `Ordered` interface. Here, it is the order of registration that dictates the order of execution. Note also that `BeanPostProcessor` instances registered programmatically are always processed before those registered through auto-detection, regardless of any explicit ordering.

NOTE

`BeanPostProcessor` *instances and AOP auto-proxying*

Classes that implement the `BeanPostProcessor` interface are special and are treated differently by the container. All `BeanPostProcessor` instances and beans that they directly reference are instantiated on startup, as part of the special startup phase of the `ApplicationContext`. Next, all `BeanPostProcessor` instances are registered in a sorted fashion and applied to all further beans in the container. Because AOP auto-proxying is implemented as a `BeanPostProcessor` itself, neither `BeanPostProcessor` instances nor the beans they directly reference are eligible for auto-proxying and, thus, do not have aspects woven into them.

For any such bean, you should see an informational log message: `Bean someBean is not eligible for getting processed by all BeanPostProcessor interfaces (for example: not eligible for auto-proxying)`.

If you have beans wired into your `BeanPostProcessor` by using autowiring or `@Resource` (which may fall back to autowiring), Spring might access unexpected beans when searching for type-matching dependency candidates and, therefore, make them ineligible for auto-proxying or other kinds of bean post-processing. For example, if you have a dependency annotated with `@Resource` where the field or setter name does not directly correspond to the declared name of a bean and no name attribute is used, Spring accesses other beans for matching them by type.

The following examples show how to write, register, and use `BeanPostProcessor` instances in an `ApplicationContext`.

Example: Hello World, `BeanPostProcessor`-style

This first example illustrates basic usage. The example shows a custom `BeanPostProcessor` implementation that invokes the `toString()` method of each bean as it is created by the container and prints the resulting string to the system console.

The following listing shows the custom `BeanPostProcessor` implementation class definition:

```java
package scripting;

import org.springframework.beans.factory.config.BeanPostProcessor;

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {

    // simply return the instantiated bean as-is
    public Object postProcessBeforeInitialization(Object bean, String beanName) {
        return bean; // we could potentially return any object reference here...
    }

    public Object postProcessAfterInitialization(Object bean, String beanName) {
        System.out.println("Bean '" + beanName + "' created : " + bean.toString()
);
        return bean;
    }
}
```

The following `beans` element uses the `InstantiationTracingBeanPostProcessor`:

```xml
<?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:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/lang
        https://www.springframework.org/schema/lang/spring-lang.xsd">

    <lang:groovy id="messenger"
            script-source=
"classpath:org/springframework/scripting/groovy/Messenger.groovy">
        <lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>
    </lang:groovy>

    <!--
    when the above bean (messenger) is instantiated, this custom
    BeanPostProcessor implementation will output the fact to the system console
    -->
    <bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>
```

Notice how the `InstantiationTracingBeanPostProcessor` is merely defined. It does not even have a name, and, because it is a bean, it can be dependency-injected as you would any other bean. (The preceding configuration also defines a bean that is backed by a Groovy script. The Spring dynamic language support is detailed in the chapter entitled Dynamic Language Support.)

The following Java application runs the preceding code and configuration:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.scripting.Messenger;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext(
"scripting/beans.xml");
        Messenger messenger = (Messenger) ctx.getBean("messenger");
        System.out.println(messenger);
    }

}
```

The output of the preceding application resembles the following:

```
Bean 'messenger' created :
org.springframework.scripting.groovy.GroovyMessenger@272961
org.springframework.scripting.groovy.GroovyMessenger@272961
```

Example: The `RequiredAnnotationBeanPostProcessor`

Using callback interfaces or annotations in conjunction with a custom `BeanPostProcessor` implementation is a common means of extending the Spring IoC container. An example is Spring's `RequiredAnnotationBeanPostProcessor` — a `BeanPostProcessor` implementation that ships with the Spring distribution and that ensures that JavaBean properties on beans that are marked with an (arbitrary) annotation are actually (configured to be) dependency-injected with a value.

Customizing Configuration Metadata with a `BeanFactoryPostProcessor`

The next extension point that we look at is the `org.springframework.beans.factory.config.BeanFactoryPostProcessor`. The semantics of this interface are similar to those of the `BeanPostProcessor`, with one major difference: `BeanFactoryPostProcessor` operates on the bean configuration metadata. That is, the Spring IoC container lets a `BeanFactoryPostProcessor` read the configuration metadata and potentially change it *before* the container instantiates any beans other than `BeanFactoryPostProcessor` instances.

You can configure multiple BeanFactoryPostProcessor instances, and you can control the order in which these BeanFactoryPostProcessor instances run by setting the order property. However, you can only set this property if the BeanFactoryPostProcessor implements the Ordered interface. If you write your own BeanFactoryPostProcessor, you should consider implementing the Ordered interface, too. See the javadoc of the {api-spring-framework}/beans/factory/config/BeanFactoryPostProcessor.html[BeanFactoryPostProcessor] and {api-spring-framework}/core/Ordered.html[Ordered] interfaces for more details.

NOTE

If you want to change the actual bean instances (that is, the objects that are created from the configuration metadata), then you instead need to use a BeanPostProcessor (described earlier in Customizing Beans by Using a BeanPostProcessor). While it is technically possible to work with bean instances within a BeanFactoryPostProcessor (for example, by using BeanFactory.getBean()), doing so causes premature bean instantiation, violating the standard container lifecycle. This may cause negative side effects, such as bypassing bean post processing.

Also, BeanFactoryPostProcessor instances are scoped per-container. This is only relevant if you use container hierarchies. If you define a BeanFactoryPostProcessor in one container, it is applied only to the bean definitions in that container. Bean definitions in one container are not post-processed by BeanFactoryPostProcessor instances in another container, even if both containers are part of the same hierarchy.

A bean factory post-processor is automatically executed when it is declared inside an ApplicationContext, in order to apply changes to the configuration metadata that define the container. Spring includes a number of predefined bean factory post-processors, such as PropertyOverrideConfigurer and PropertyPlaceholderConfigurer. You can also use a custom BeanFactoryPostProcessor — for example, to register custom property editors.

An ApplicationContext automatically detects any beans that are deployed into it that implement the BeanFactoryPostProcessor interface. It uses these beans as bean factory post-processors, at the appropriate time. You can deploy these post-processor beans as you would any other bean.

NOTE

As with BeanPostProcessors , you typically do not want to configure BeanFactoryPostProcessors for lazy initialization. If no other bean references a Bean(Factory)PostProcessor, that post-processor will not get instantiated at all. Thus, marking it for lazy initialization will be ignored, and the Bean(Factory)PostProcessor will be instantiated eagerly even if you set the default-lazy-init attribute to true on the declaration of your <beans /> element.

Example: The Class Name Substitution PropertyPlaceholderConfigurer

You can use the PropertyPlaceholderConfigurer to externalize property values from a bean definition in a separate file by using the standard Java Properties format. Doing so enables the person deploying an application to customize environment-specific properties, such as database URLs and passwords, without the complexity or risk of modifying the main XML definition file or files for the container.

Consider the following XML-based configuration metadata fragment, where a `DataSource` with placeholder values is defined:

```
<bean class=
"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations" value="classpath:com/something/jdbc.properties"/>
</bean>

<bean id="dataSource" destroy-method="close"
        class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

The example shows properties configured from an external `Properties` file. At runtime, a `PropertyPlaceholderConfigurer` is applied to the metadata that replaces some properties of the DataSource. The values to replace are specified as placeholders of the form `${property-name}`, which follows the Ant and log4j and JSP EL style.

The actual values come from another file in the standard Java `Properties` format:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

Therefore, the `${jdbc.username}` string is replaced at runtime with the value, 'sa', and the same applies for other placeholder values that match keys in the properties file. The `PropertyPlaceholderConfigurer` checks for placeholders in most properties and attributes of a bean definition. Furthermore, you can customize the placeholder prefix and suffix.

With the `context` namespace introduced in Spring 2.5, you can configure property placeholders with a dedicated configuration element. You can provide one or more locations as a comma-separated list in the `location` attribute, as the following example shows:

```
<context:property-placeholder location="classpath:com/something/jdbc.properties"/>
```

The `PropertyPlaceholderConfigurer` not only looks for properties in the `Properties` file you specify. By default, if it cannot find a property in the specified properties files, it also checks against the Java `System` properties. You can customize this behavior by setting the `systemPropertiesMode` property of the configurer with one of the following three supported integer values:

- `never` (0): Never check system properties.
- `fallback` (1): Check system properties if not resolvable in the specified properties files. This is the default.
- `override` (2): Check system properties first, before trying the specified properties files. This lets system properties override any other property source.

See the {api-spring-framework}/beans/factory/config/PropertyPlaceholderConfigurer.html[`PropertyPlaceholderConfigurer`] javadoc for more information.

TIP

You can use the `PropertyPlaceholderConfigurer` to substitute class names, which is sometimes useful when you have to pick a particular implementation class at runtime. The following example shows how to do so:

```
<bean class=
"org.springframework.beans.factory.config.PropertyPlaceholderConfigure
r">
    <property name="locations">
        <value>classpath:com/something/strategy.properties</value>
    </property>
    <property name="properties">
        <value>
custom.strategy.class=com.something.DefaultStrategy</value>
    </property>
</bean>

<bean id="serviceStrategy" class="${custom.strategy.class}"/>
```

If the class cannot be resolved at runtime to a valid class, resolution of the bean fails when it is about to be created, which is during the `preInstantiateSingletons()` phase of an `ApplicationContext` for a non-lazy-init bean.

Example: The `PropertyOverrideConfigurer`

The `PropertyOverrideConfigurer`, another bean factory post-processor, resembles the `PropertyPlaceholderConfigurer`, but unlike the latter, the original definitions can have default values or no values at all for bean properties. If an overriding `Properties` file does not have an entry for a certain bean property, the default context definition is used.

Note that the bean definition is not aware of being overridden, so it is not immediately obvious from the XML definition file that the override configurer is being used. In case of multiple `PropertyOverrideConfigurer` instances that define different values for the same bean property, the last one wins, due to the overriding mechanism.

Properties file configuration lines take the following format:

```
beanName.property=value
```

The following listing shows an example of the format:

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql:mydb
```

This example file can be used with a container definition that contains a bean called `dataSource` that has `driver` and `url` properties.

Compound property names are also supported, as long as every component of the path except the final property being overridden is already non-null (presumably initialized by the constructors). In the following example, the `sammy` property of the `bob` property of the `fred` property of the `tom` bean is set to the scalar value `123`:

```
tom.fred.bob.sammy=123
```

NOTE	Specified override values are always literal values. They are not translated into bean references. This convention also applies when the original value in the XML bean definition specifies a bean reference.

With the `context` namespace introduced in Spring 2.5, it is possible to configure property overriding with a dedicated configuration element, as the following example shows:

```
<context:property-override location="classpath:override.properties"/>
```

Customizing Instantiation Logic with a `FactoryBean`

You can implement the `org.springframework.beans.factory.FactoryBean` interface for objects that are themselves factories.

The `FactoryBean` interface is a point of pluggability into the Spring IoC container's instantiation logic. If you have complex initialization code that is better expressed in Java as opposed to a (potentially) verbose amount of XML, you can create your own `FactoryBean`, write the complex initialization inside that class, and then plug your custom `FactoryBean` into the container.

The `FactoryBean` interface provides three methods:

- `Object getObject()`: Returns an instance of the object this factory creates. The instance can possibly be shared, depending on whether this factory returns singletons or prototypes.
- `boolean isSingleton()`: Returns `true` if this `FactoryBean` returns singletons or `false` otherwise.
- `Class getObjectType()`: Returns the object type returned by the `getObject()` method or `null` if the type is not known in advance.

The `FactoryBean` concept and interface is used in a number of places within the Spring Framework. More than 50 implementations of the `FactoryBean` interface ship with Spring itself.

When you need to ask a container for an actual `FactoryBean` instance itself instead of the bean it produces, preface the bean's `id` with the ampersand symbol (`&`) when calling the `getBean()` method of the `ApplicationContext`. So, for a given `FactoryBean` with an `id` of `myBean`, invoking `getBean("myBean")` on the container returns the product of the `FactoryBean`, whereas invoking `getBean("&myBean")` returns the `FactoryBean` instance itself.

Annotation-based Container Configuration

Are annotations better than XML for configuring Spring?

The introduction of annotation-based configuration raised the question of whether this approach is "better" than XML. The short answer is "it depends." The long answer is that each approach has its pros and cons, and, usually, it is up to the developer to decide which strategy suits them better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.

No matter the choice, Spring can accommodate both styles and even mix them together. It is worth pointing out that through its JavaConfig option, Spring lets annotations be used in a non-invasive way, without touching the target components source code and that, in terms of tooling, all configuration styles are supported by the Spring Tool Suite.

An alternative to XML setup is provided by annotation-based configuration, which relies on the bytecode metadata for wiring up components instead of angle-bracket declarations. Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration. As mentioned in Example: The `RequiredAnnotationBeanPostProcessor`, using a `BeanPostProcessor` in conjunction with annotations is a common means of extending the Spring IoC container. For example, Spring 2.0 introduced the possibility of enforcing required properties with the `@Required` annotation. Spring 2.5 made it possible to follow that same general approach to drive Spring's dependency injection. Essentially, the `@Autowired` annotation provides the same capabilities as described in Autowiring Collaborators but with more fine-grained control and wider applicability. Spring 2.5 also added support for JSR-250 annotations, such as `@PostConstruct` and `@PreDestroy`. Spring 3.0 added support for JSR-330 (Dependency Injection for Java) annotations contained in the `javax.inject` package such as `@Inject` and `@Named`. Details about those annotations can be found in the relevant section.

NOTE	Annotation injection is performed before XML injection. Thus, the XML configuration overrides the annotations for properties wired through both approaches.

As always, you can register them as individual bean definitions, but they can also be implicitly registered by including the following tag in an XML-based Spring configuration (notice the inclusion of the `context` namespace):

```
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

</beans>
```

(The implicitly registered post-processors include {api-spring-framework}/beans/factory/annotation/AutowiredAnnotationBeanPostProcessor.html[`AutowiredAnnotationBeanPostProcessor`], {api-spring-framework}/context/annotation/CommonAnnotationBeanPostProcessor.html[`CommonAnnotationBeanPostProcessor`], {api-spring-framework}/orm/jpa/support/PersistenceAnnotationBeanPostProcessor.html[`PersistenceAnnotationBeanPostProcessor`], and the aforementioned {api-spring-framework}/beans/factory/annotation/RequiredAnnotationBeanPostProcessor.html[`RequiredAnnotationBeanPostProcessor`].)

NOTE	`<context:annotation-config/>` only looks for annotations on beans in the same application context in which it is defined. This means that, if you put `<context:annotation-config/>` in a `WebApplicationContext` for a `DispatcherServlet`, it only checks for `@Autowired` beans in your controllers, and not your services. See The DispatcherServlet for more information.

@Required

The `@Required` annotation applies to bean property setter methods, as in the following example:

```java
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Required
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

This annotation indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring. The container throws an exception if the affected bean property has not been populated. This allows for eager and explicit failure, avoiding `NullPointerException` instances or the like later on. We still recommend that you put assertions into the bean class itself (for example, into an init method). Doing so enforces those required references and values even when you use the class outside of a container.

NOTE	The `@Required` annotation is formally deprecated as of Spring Framework 5.1, in favor of using constructor injection for required settings (or a custom implementation of `InitializingBean.afterPropertiesSet()` along with bean property setter methods).

Using `@Autowired`

NOTE	JSR 330's `@Inject` annotation can be used in place of Spring's `@Autowired` annotation in the examples included in this section. See here for more details.

You can apply the `@Autowired` annotation to constructors, as the following example shows:

```java
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

NOTE As of Spring Framework 4.3, an `@Autowired` annotation on such a constructor is no longer necessary if the target bean defines only one constructor to begin with. However, if several constructors are available, at least one must be annotated to teach the container which one to use.

You can also apply the `@Autowired` annotation to “traditional” setter methods, as the following example shows:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

You can also apply the annotation to methods with arbitrary names and multiple arguments, as the following example shows:

```
public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(MovieCatalog movieCatalog,
            CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

You can apply `@Autowired` to fields as well and even mix it with constructors, as the following example shows:

```java
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    private MovieCatalog movieCatalog;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

TIP

Make sure that your target components (for example, `MovieCatalog` or `CustomerPreferenceDao`) are consistently declared by the type that you use for your `@Autowired`-annotated injection points. Otherwise, injection may fail due to no type match found at runtime.

For XML-defined beans or component classes found through a classpath scan, the container usually knows the concrete type up front. However, for `@Bean` factory methods, you need to make sure that the declared return type is sufficiently expressive. For components that implement several interfaces or for components potentially referred to by their implementation type, consider declaring the most specific return type on your factory method (at least as specific as required by the injection points referring to your bean).

You can also provide all beans of a particular type from the `ApplicationContext` by adding the annotation to a field or method that expects an array of that type, as the following example shows:

```java
public class MovieRecommender {

    @Autowired
    private MovieCatalog[] movieCatalogs;

    // ...
}
```

The same applies for typed collections, as the following example shows:

```java
public class MovieRecommender {

    private Set<MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Set<MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }

    // ...
}
```

TIP

Your target beans can implement the `org.springframework.core.Ordered` interface or use the `@Order` or standard `@Priority` annotation if you want items in the array or list to be sorted in a specific order. Otherwise, their order follows the registration order of the corresponding target bean definitions in the container.

You can declare the `@Order` annotation at the target class level and on `@Bean` methods, potentially by individual bean definition (in case of multiple definitions that use the same bean class). `@Order` values may influence priorities at injection points, but be aware that they do not influence singleton startup order, which is an orthogonal concern determined by dependency relationships and `@DependsOn` declarations.

Note that the standard `javax.annotation.Priority` annotation is not available at the `@Bean` level, since it cannot be declared on methods. Its semantics can be modeled through `@Order` values in combination with `@Primary` on a single bean for each type.

Even typed `Map` instances can be autowired as long as the expected key type is `String`. The Map values contain all beans of the expected type, and the keys contain the corresponding bean names, as the following example shows:

```java
public class MovieRecommender {

    private Map<String, MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }

    // ...
}
```

By default, autowiring fails when no matching candidate beans are available for a given injection point. In the case of a declared array, collection or map, at least one matching element is expected.

The default behavior is to treat annotated methods and fields as indicating required dependencies. You can change this behavior as demonstrated in the following example, enabling the framework to skip a non-satisfiable injection point through marking it as non-required:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired(required = false)
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

A non-required method will not be called at all if its dependency (or one of its dependencies in case of multiple arguments) is not available. A non-required field will not get populated at all in such case, leaving its default value in place.

Injected constructor and factory method arguments are a special case since the 'required' flag on `@Autowired` has a somewhat different meaning due to Spring's constructor resolution algorithm potentially dealing with multiple constructors. Constructor and factory method arguments are effectively required by default but with a few special rules in a single-constructor scenario, such as multi-element injection points (arrays, collections, maps) resolving to empty instances if no matching beans are available. This allows for a common implementation pattern where all dependencies can be declared in a unique multi-argument constructor, e.g. declared as a single public constructor without an `@Autowired` annotation.

NOTE

Only one annotated constructor per class can be marked as required, but multiple non-required constructors can be annotated. In that case, each is considered among the candidates and Spring uses the greediest constructor whose dependencies can be satisfied — that is, the constructor that has the largest number of arguments. The constructor resolution algorithm is the same as for non-annotated classes with overloaded constructors, just narrowing the candidates to annotated constructors.

The 'required' attribute of `@Autowired` is recommended over the `@Required` annotation on setter methods. The 'required' attribute indicates that the property is not required for autowiring purposes. The property is ignored if it cannot be autowired. `@Required`, on the other hand, is stronger in that it enforces the property to be set by any means supported by the container. If no value is defined, a corresponding exception is raised.

Alternatively, you can express the non-required nature of a particular dependency through Java 8's `java.util.Optional`, as the following example shows:

```java
public class SimpleMovieLister {

    @Autowired
    public void setMovieFinder(Optional<MovieFinder> movieFinder) {
        ...
    }
}
```

As of Spring Framework 5.0, you can also use a `@Nullable` annotation (of any kind in any package — for example, `javax.annotation.Nullable` from JSR-305):

```java
public class SimpleMovieLister {

    @Autowired
    public void setMovieFinder(@Nullable MovieFinder movieFinder) {
        ...
    }
}
```

You can also use `@Autowired` for interfaces that are well-known resolvable dependencies: `BeanFactory`, `ApplicationContext`, `Environment`, `ResourceLoader`, `ApplicationEventPublisher`, and `MessageSource`. These interfaces and their extended interfaces, such as `ConfigurableApplicationContext` or `ResourcePatternResolver`, are automatically resolved, with no special setup necessary. The following example autowires an `ApplicationContext` object:

```java
public class MovieRecommender {

    @Autowired
    private ApplicationContext context;

    public MovieRecommender() {
    }

    // ...
}
```

NOTE The `@Autowired`, `@Inject`, `@Value`, and `@Resource` annotations are handled by Spring `BeanPostProcessor` implementations. This means that you cannot apply these annotations within your own `BeanPostProcessor` or `BeanFactoryPostProcessor` types (if any). These types must be 'wired up' explicitly by using XML or a Spring `@Bean` method.

Fine-tuning Annotation-based Autowiring with `@Primary`

Because autowiring by type may lead to multiple candidates, it is often necessary to have more control over the selection process. One way to accomplish this is with Spring's `@Primary` annotation. `@Primary` indicates that a particular bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency. If exactly one primary bean exists among the candidates, it becomes the autowired value.

Consider the following configuration that defines `firstMovieCatalog` as the primary `MovieCatalog`:

```java
@Configuration
public class MovieConfiguration {

    @Bean
    @Primary
    public MovieCatalog firstMovieCatalog() { ... }

    @Bean
    public MovieCatalog secondMovieCatalog() { ... }

    // ...
}
```

With the preceding configuration, the following `MovieRecommender` is autowired with the `firstMovieCatalog`:

```java
public class MovieRecommender {

    @Autowired
    private MovieCatalog movieCatalog;

    // ...
}
```

The corresponding bean definitions follow:

```xml
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog" primary="true">
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

Fine-tuning Annotation-based Autowiring with Qualifiers

`@Primary` is an effective way to use autowiring by type with several instances when one primary candidate can be determined. When you need more control over the selection process, you can use Spring's `@Qualifier` annotation. You can associate qualifier values with specific arguments, narrowing the set of type matches so that a specific bean is chosen for each argument. In the simplest case, this can be a plain descriptive value, as shown in the following example:

```java
public class MovieRecommender {

    @Autowired
    @Qualifier("main")
    private MovieCatalog movieCatalog;

    // ...
}
```

You can also specify the `@Qualifier` annotation on individual constructor arguments or method parameters, as shown in the following example:

```java
public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public void prepare(@Qualifier("main") MovieCatalog movieCatalog,
            CustomerPreferenceDao customerPreferenceDao) {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```

The following example shows corresponding bean definitions.

```xml
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="main"/> ①

        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier value="action"/> ②

        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

① The bean with the `main` qualifier value is wired with the constructor argument that is qualified with the same value.

② The bean with the `action` qualifier value is wired with the constructor argument that is qualified with the same value.

For a fallback match, the bean name is considered a default qualifier value. Thus, you can define the bean with an `id` of `main` instead of the nested qualifier element, leading to the same matching result. However, although you can use this convention to refer to specific beans by name, `@Autowired` is fundamentally about type-driven injection with optional semantic qualifiers. This means that qualifier values, even with the bean name fallback, always have narrowing semantics within the set of type matches. They do not semantically express a reference to a unique bean `id`. Good qualifier values are `main` or `EMEA` or `persistent`, expressing characteristics of a specific component that are independent from the bean `id`, which may be auto-generated in case of an anonymous bean definition such as the one in the preceding example.

Qualifiers also apply to typed collections, as discussed earlier — for example, to `Set<MovieCatalog>`. In this case, all matching beans, according to the declared qualifiers, are injected as a collection. This implies that qualifiers do not have to be unique. Rather, they constitute filtering criteria. For example, you can define multiple `MovieCatalog` beans with the same qualifier value “action”, all of which are injected into a `Set<MovieCatalog>` annotated with `@Qualifier("action")`.

TIP

Letting qualifier values select against target bean names, within the type-matching candidates, does not require a `@Qualifier` annotation at the injection point. If there is no other resolution indicator (such as a qualifier or a primary marker), for a non-unique dependency situation, Spring matches the injection point name (that is, the field name or parameter name) against the target bean names and choose the same-named candidate, if any.

That said, if you intend to express annotation-driven injection by name, do not primarily use `@Autowired`, even if it is capable of selecting by bean name among type-matching candidates. Instead, use the JSR-250 `@Resource` annotation, which is semantically defined to identify a specific target component by its unique name, with the declared type being irrelevant for the matching process. `@Autowired` has rather different semantics: After selecting candidate beans by type, the specified `String` qualifier value is considered within those type-selected candidates only (for example, matching an `account` qualifier against beans marked with the same qualifier label).

For beans that are themselves defined as a collection, `Map`, or array type, `@Resource` is a fine solution, referring to the specific collection or array bean by unique name. That said, as of 4.3, collection, you can match `Map`, and array types through Spring's `@Autowired` type matching algorithm as well, as long as the element type information is preserved in `@Bean` return type signatures or collection inheritance hierarchies. In this case, you can use qualifier values to select among same-typed collections, as outlined in the previous paragraph.

As of 4.3, `@Autowired` also considers self references for injection (that is, references back to the bean that is currently injected). Note that self injection is a fallback. Regular dependencies on other components always have precedence. In that sense, self references do not participate in regular candidate selection and are therefore in particular never primary. On the contrary, they always end up as lowest precedence. In practice, you should use self references as a last resort only (for example, for calling other methods on the same instance through the bean's transactional proxy). Consider factoring out the effected methods to a separate delegate bean in such a scenario. Alternatively, you can use `@Resource`, which may obtain a proxy back to the current bean by its unique name.

`@Autowired` applies to fields, constructors, and multi-argument methods, allowing for narrowing through qualifier annotations at the parameter level. By contrast, `@Resource` is supported only for fields and bean property setter methods with a single argument. As a consequence, you should stick with qualifiers if your injection target is a constructor or a multi-argument method.

You can create your own custom qualifier annotations. To do so, define an annotation and provide the `@Qualifier` annotation within your definition, as the following example shows:

```java
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Genre {

    String value();
}
```

Then you can provide the custom qualifier on autowired fields and parameters, as the following example shows:

```java
public class MovieRecommender {

    @Autowired
    @Genre("Action")
    private MovieCatalog actionCatalog;

    private MovieCatalog comedyCatalog;

    @Autowired
    public void setComedyCatalog(@Genre("Comedy") MovieCatalog comedyCatalog) {
        this.comedyCatalog = comedyCatalog;
    }

    // ...
}
```

Next, you can provide the information for the candidate bean definitions. You can add `<qualifier/>` tags as sub-elements of the `<bean/>` tag and then specify the `type` and `value` to match your custom qualifier annotations. The type is matched against the fully-qualified class name of the annotation. Alternately, as a convenience if no risk of conflicting names exists, you can use the short class name. The following example demonstrates both approaches:

```xml
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="Genre" value="Action"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="example.Genre" value="Comedy"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean id="movieRecommender" class="example.MovieRecommender"/>

</beans>
```

In Classpath Scanning and Managed Components, you can see an annotation-based alternative to providing the qualifier metadata in XML. Specifically, see Providing Qualifier Metadata with Annotations.

In some cases, using an annotation without a value may suffice. This can be useful when the annotation serves a more generic purpose and can be applied across several different types of dependencies. For example, you may provide an offline catalog that can be searched when no Internet connection is available. First, define the simple annotation, as the following example shows:

```java
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Offline {

}
```

Then add the annotation to the field or property to be autowired, as shown in the following example:

```java
public class MovieRecommender {

    @Autowired
    @Offline ①
    private MovieCatalog offlineCatalog;

    // ...
}
```

① This line adds the `@Offline` annotation.

Now the bean definition only needs a qualifier `type`, as shown in the following example:

```xml
<bean class="example.SimpleMovieCatalog">
    <qualifier type="Offline"/> ①
    <!-- inject any dependencies required by this bean -->
</bean>
```

① This element specifies the qualifier.

You can also define custom qualifier annotations that accept named attributes in addition to or instead of the simple `value` attribute. If multiple attribute values are then specified on a field or parameter to be autowired, a bean definition must match all such attribute values to be considered an autowire candidate. As an example, consider the following annotation definition:

```java
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MovieQualifier {

    String genre();

    Format format();
}
```

In this case `Format` is an enum, defined as follows:

```java
public enum Format {
    VHS, DVD, BLURAY
}
```

The fields to be autowired are annotated with the custom qualifier and include values for both attributes: `genre` and `format`, as the following example shows:

```
public class MovieRecommender {

    @Autowired
    @MovieQualifier(format=Format.VHS, genre="Action")
    private MovieCatalog actionVhsCatalog;

    @Autowired
    @MovieQualifier(format=Format.VHS, genre="Comedy")
    private MovieCatalog comedyVhsCatalog;

    @Autowired
    @MovieQualifier(format=Format.DVD, genre="Action")
    private MovieCatalog actionDvdCatalog;

    @Autowired
    @MovieQualifier(format=Format.BLURAY, genre="Comedy")
    private MovieCatalog comedyBluRayCatalog;

    // ...
}
```

Finally, the bean definitions should contain matching qualifier values. This example also demonstrates that you can use bean meta attributes instead of the `<qualifier/>` elements. If available, the `<qualifier/>` element and its attributes take precedence, but the autowiring mechanism falls back on the values provided within the `<meta/>` tags if no such qualifier is present, as in the last two bean definitions in the following example:

```xml
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="MovieQualifier">
            <attribute key="format" value="VHS"/>
            <attribute key="genre" value="Action"/>
        </qualifier>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <qualifier type="MovieQualifier">
            <attribute key="format" value="VHS"/>
            <attribute key="genre" value="Comedy"/>
        </qualifier>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <meta key="format" value="DVD"/>
        <meta key="genre" value="Action"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

    <bean class="example.SimpleMovieCatalog">
        <meta key="format" value="BLURAY"/>
        <meta key="genre" value="Comedy"/>
        <!-- inject any dependencies required by this bean -->
    </bean>

</beans>
```

Using Generics as Autowiring Qualifiers

In addition to the `@Qualifier` annotation, you can use Java generic types as an implicit form of qualification. For example, suppose you have the following configuration:

```java
@Configuration
public class MyConfiguration {

    @Bean
    public StringStore stringStore() {
        return new StringStore();
    }

    @Bean
    public IntegerStore integerStore() {
        return new IntegerStore();
    }
}
```

Assuming that the preceding beans implement a generic interface, (that is, `Store<String>` and `Store<Integer>`), you can `@Autowire` the `Store` interface and the generic is used as a qualifier, as the following example shows:

```java
@Autowired
private Store<String> s1; // <String> qualifier, injects the stringStore bean

@Autowired
private Store<Integer> s2; // <Integer> qualifier, injects the integerStore bean
```

Generic qualifiers also apply when autowiring lists, `Map` instances and arrays. The following example autowires a generic `List`:

```java
// Inject all Store beans as long as they have an <Integer> generic
// Store<String> beans will not appear in this list
@Autowired
private List<Store<Integer>> s;
```

Using `CustomAutowireConfigurer`

{api-spring-framework}/beans/factory/annotation/CustomAutowireConfigurer.html[`CustomAutowireConfigurer`] is a `BeanFactoryPostProcessor` that lets you register your own custom qualifier annotation types, even if they are not annotated with Spring's `@Qualifier` annotation. The following example shows how to use `CustomAutowireConfigurer`:

```
<bean id="customAutowireConfigurer"
        class=
"org.springframework.beans.factory.annotation.CustomAutowireConfigurer">
    <property name="customQualifierTypes">
        <set>
            <value>example.CustomQualifier</value>
        </set>
    </property>
</bean>
```

The `AutowireCandidateResolver` determines autowire candidates by:

- The `autowire-candidate` value of each bean definition
- Any `default-autowire-candidates` patterns available on the `<beans/>` element
- The presence of `@Qualifier` annotations and any custom annotations registered with the `CustomAutowireConfigurer`

When multiple beans qualify as autowire candidates, the determination of a “primary” is as follows: If exactly one bean definition among the candidates has a `primary` attribute set to `true`, it is selected.

Injection with `@Resource`

Spring also supports injection by using the JSR-250 `@Resource` annotation (`javax.annotation.Resource`) on fields or bean property setter methods. This is a common pattern in Java EE: for example, in JSF-managed beans and JAX-WS endpoints. Spring supports this pattern for Spring-managed objects as well.

`@Resource` takes a name attribute. By default, Spring interprets that value as the bean name to be injected. In other words, it follows by-name semantics, as demonstrated in the following example:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Resource(name="myMovieFinder") ①
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

① This line injects a `@Resource`.

If no name is explicitly specified, the default name is derived from the field name or setter method.

In case of a field, it takes the field name. In case of a setter method, it takes the bean property name. The following example is going to have the bean named `movieFinder` injected into its setter method:

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Resource
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

NOTE The name provided with the annotation is resolved as a bean name by the `ApplicationContext` of which the `CommonAnnotationBeanPostProcessor` is aware. The names can be resolved through JNDI if you configure Spring's {api-spring-framework}/jndi/support/SimpleJndiBeanFactory.html[`SimpleJndiBeanFactory`] explicitly. However, we recommend that you rely on the default behavior and use Spring's JNDI lookup capabilities to preserve the level of indirection.

In the exclusive case of `@Resource` usage with no explicit name specified, and similar to `@Autowired`, `@Resource` finds a primary type match instead of a specific named bean and resolves well known resolvable dependencies: the `BeanFactory`, `ApplicationContext`, `ResourceLoader`, `ApplicationEventPublisher`, and `MessageSource` interfaces.

Thus, in the following example, the `customerPreferenceDao` field first looks for a bean named "customerPreferenceDao" and then falls back to a primary type match for the type `CustomerPreferenceDao`:

```java
public class MovieRecommender {

    @Resource
    private CustomerPreferenceDao customerPreferenceDao;

    @Resource
    private ApplicationContext context; ①

    public MovieRecommender() {
    }

    // ...
}
```

① The `context` field is injected based on the known resolvable dependency type: `ApplicationContext`.

Using `@PostConstruct` and `@PreDestroy`

The `CommonAnnotationBeanPostProcessor` not only recognizes the `@Resource` annotation but also the JSR-250 lifecycle annotations: `javax.annotation.PostConstruct` and `javax.annotation.PreDestroy`. Introduced in Spring 2.5, the support for these annotations offers an alternative to the lifecycle callback mechanism described in initialization callbacks and destruction callbacks. Provided that the `CommonAnnotationBeanPostProcessor` is registered within the Spring `ApplicationContext`, a method carrying one of these annotations is invoked at the same point in the lifecycle as the corresponding Spring lifecycle interface method or explicitly declared callback method. In the following example, the cache is pre-populated upon initialization and cleared upon destruction:

```java
public class CachingMovieLister {

    @PostConstruct
    public void populateMovieCache() {
        // populates the movie cache upon initialization...
    }

    @PreDestroy
    public void clearMovieCache() {
        // clears the movie cache upon destruction...
    }
}
```

For details about the effects of combining various lifecycle mechanisms, see Combining Lifecycle Mechanisms.

NOTE	Like `@Resource`, the `@PostConstruct` and `@PreDestroy` annotation types were a part of the standard Java libraries from JDK 6 to 8. However, the entire `javax.annotation` package got separated from the core Java modules in JDK 9 and eventually removed in JDK 11. If needed, the `javax.annotation-api` artifact needs to be obtained via Maven Central now, simply to be added to the application's classpath like any other library.

Classpath Scanning and Managed Components

Most examples in this chapter use XML to specify the configuration metadata that produces each `BeanDefinition` within the Spring container. The previous section (Annotation-based Container Configuration) demonstrates how to provide a lot of the configuration metadata through source-level annotations. Even in those examples, however, the "base" bean definitions are explicitly defined in the XML file, while the annotations drive only the dependency injection. This section describes an option for implicitly detecting the candidate components by scanning the classpath. Candidate components are classes that match against a filter criteria and have a corresponding bean definition registered with the container. This removes the need to use XML to perform bean registration. Instead, you can use annotations (for example, `@Component`), AspectJ type expressions, or your own custom filter criteria to select which classes have bean definitions registered with the container.

NOTE	Starting with Spring 3.0, many features provided by the Spring JavaConfig project are part of the core Spring Framework. This allows you to define beans using Java rather than using the traditional XML files. Take a look at the `@Configuration`, `@Bean`, `@Import`, and `@DependsOn` annotations for examples of how to use these new features.

`@Component` and Further Stereotype Annotations

The `@Repository` annotation is a marker for any class that fulfills the role or stereotype of a repository (also known as Data Access Object or DAO). Among the uses of this marker is the automatic translation of exceptions, as described in Exception Translation.

Spring provides further stereotype annotations: `@Component`, `@Service`, and `@Controller`. `@Component` is a generic stereotype for any Spring-managed component. `@Repository`, `@Service`, and `@Controller` are specializations of `@Component` for more specific use cases (in the persistence, service, and presentation layers, respectively). Therefore, you can annotate your component classes with `@Component`, but, by annotating them with `@Repository`, `@Service`, or `@Controller` instead, your classes are more properly suited for processing by tools or associating with aspects. For example, these stereotype annotations make ideal targets for pointcuts. `@Repository`, `@Service`, and `@Controller` can also carry additional semantics in future releases of the Spring Framework. Thus, if you are choosing between using `@Component` or `@Service` for your service layer, `@Service` is clearly the better choice. Similarly, as stated earlier, `@Repository` is already supported as a marker for automatic exception translation in your persistence layer.

Using Meta-annotations and Composed Annotations

Many of the annotations provided by Spring can be used as meta-annotations in your own code. A meta-annotation is an annotation that can be applied to another annotation. For example, the `@Service` annotation mentioned earlier is meta-annotated with `@Component`, as the following example shows:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component ①
public @interface Service {

    // ....
}
```

① The `Component` causes `@Service` to be treated in the same way as `@Component`.

You can also combine meta-annotations to create "composed annotations". For example, the `@RestController` annotation from Spring MVC is composed of `@Controller` and `@ResponseBody`.

In addition, composed annotations can optionally redeclare attributes from meta-annotations to allow customization. This can be particularly useful when you want to only expose a subset of the meta-annotation's attributes. For example, Spring's `@SessionScope` annotation hardcodes the scope name to `session` but still allows customization of the `proxyMode`. The following listing shows the definition of the `SessionScope` annotation:

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Scope(WebApplicationContext.SCOPE_SESSION)
public @interface SessionScope {

    /**
     * Alias for {@link Scope#proxyMode}.
     * <p>Defaults to {@link ScopedProxyMode#TARGET_CLASS}.
     */
    @AliasFor(annotation = Scope.class)
    ScopedProxyMode proxyMode() default ScopedProxyMode.TARGET_CLASS;

}
```

You can then use `@SessionScope` without declaring the `proxyMode` as follows:

```java
@Service
@SessionScope
public class SessionScopedService {
    // ...
}
```

You can also override the value for the `proxyMode`, as the following example shows:

```java
@Service
@SessionScope(proxyMode = ScopedProxyMode.INTERFACES)
public class SessionScopedUserService implements UserService {
    // ...
}
```

For further details, see the Spring Annotation Programming Model wiki page.

Automatically Detecting Classes and Registering Bean Definitions

Spring can automatically detect stereotyped classes and register corresponding `BeanDefinition` instances with the `ApplicationContext`. For example, the following two classes are eligible for such autodetection:

```java
@Service
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Autowired
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
```

```java
@Repository
public class JpaMovieFinder implements MovieFinder {
    // implementation elided for clarity
}
```

To autodetect these classes and register the corresponding beans, you need to add `@ComponentScan` to your `@Configuration` class, where the `basePackages` attribute is a common parent package for the two classes. (Alternatively, you can specify a comma- or semicolon- or space-separated list that includes the parent package of each class.)

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    ...
}
```

NOTE For brevity, the preceding example could have used the `value` attribute of the annotation (that is, `@ComponentScan("org.example")`).

The following alternative uses XML:

```
<?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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.example"/>

</beans>
```

TIP The use of `<context:component-scan>` implicitly enables the functionality of `<context:annotation-config>`. There is usually no need to include the `<context:annotation-config>` element when using `<context:component-scan>`.

NOTE	The scanning of classpath packages requires the presence of corresponding directory entries in the classpath. When you build JARs with Ant, make sure that you do not activate the files-only switch of the JAR task. Also, classpath directories may not be exposed based on security policies in some environments — for example, standalone apps on JDK 1.7.0_45 and higher (which requires 'Trusted-Library' setup in your manifests — see https://stackoverflow.com/questions/19394570/java-jre-7u45-breaks-classloader-getresources). On JDK 9's module path (Jigsaw), Spring's classpath scanning generally works as expected. However, make sure that your component classes are exported in your `module-info` descriptors. If you expect Spring to invoke non-public members of your classes, make sure that they are 'opened' (that is, that they use an `opens` declaration instead of an `exports` declaration in your `module-info` descriptor).

Furthermore, the `AutowiredAnnotationBeanPostProcessor` and `CommonAnnotationBeanPostProcessor` are both implicitly included when you use the component-scan element. That means that the two components are autodetected and wired together — all without any bean configuration metadata provided in XML.

NOTE	You can disable the registration of `AutowiredAnnotationBeanPostProcessor` and `CommonAnnotationBeanPostProcessor` by including the annotation-config attribute with a value of `false`.

Using Filters to Customize Scanning

By default, classes annotated with `@Component`, `@Repository`, `@Service`, `@Controller`, or a custom annotation that itself is annotated with `@Component` are the only detected candidate components. However, you can modify and extend this behavior by applying custom filters. Add them as `includeFilters` or `excludeFilters` parameters of the `@ComponentScan` annotation (or as `include-filter` or `exclude-filter` child elements of the `component-scan` element). Each filter element requires the `type` and `expression` attributes. The following table describes the filtering options:

Table 5. Filter Types

Filter Type	Example Expression	Description
annotation (default)	`org.example.SomeAnnotation`	An annotation to be present at the type level in target components.
assignable	`org.example.SomeClass`	A class (or interface) that the target components are assignable to (extend or implement).
aspectj	`org.example..*Service+`	An AspectJ type expression to be matched by the target components.

Filter Type	Example Expression	Description
regex	`org\.example\.Default.*`	A regex expression to be matched by the target components class names.
custom	`org.example.MyTypeFilter`	A custom implementation of the `org.springframework.core.type.TypeFilter` interface.

The following example shows the configuration ignoring all `@Repository` annotations and using "stub" repositories instead:

```
@Configuration
@ComponentScan(basePackages = "org.example",
        includeFilters = @Filter(type = FilterType.REGEX, pattern =
".*Stub.*Repository"),
        excludeFilters = @Filter(Repository.class))
public class AppConfig {
    ...
}
```

The following listing shows the equivalent XML:

```
<beans>
    <context:component-scan base-package="org.example">
        <context:include-filter type="regex"
                expression=".*Stub.*Repository"/>
        <context:exclude-filter type="annotation"
                expression="org.springframework.stereotype.Repository"/>
    </context:component-scan>
</beans>
```

NOTE

You can also disable the default filters by setting `useDefaultFilters=false` on the annotation or by providing `use-default-filters="false"` as an attribute of the `<component-scan/>` element. This, in effect, disables automatic detection of classes annotated with `@Component`, `@Repository`, `@Service`, `@Controller`, or `@Configuration`.

Defining Bean Metadata within Components

Spring components can also contribute bean definition metadata to the container. You can do this with the same `@Bean` annotation used to define bean metadata within `@Configuration` annotated classes. The following example shows how to do so:

```java
@Component
public class FactoryMethodComponent {

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    public void doWork() {
        // Component method implementation omitted
    }
}
```

The preceding class is a Spring component that has application-specific code in its `doWork()` method. However, it also contributes a bean definition that has a factory method referring to the method `publicInstance()`. The `@Bean` annotation identifies the factory method and other bean definition properties, such as a qualifier value through the `@Qualifier` annotation. Other method-level annotations that can be specified are `@Scope`, `@Lazy`, and custom qualifier annotations.

TIP In addition to its role for component initialization, you can also place the `@Lazy` annotation on injection points marked with `@Autowired` or `@Inject`. In this context, it leads to the injection of a lazy-resolution proxy.

Autowired fields and methods are supported, as previously discussed, with additional support for autowiring of `@Bean` methods. The following example shows how to do so:

```
@Component
public class FactoryMethodComponent {

    private static int i;

    @Bean
    @Qualifier("public")
    public TestBean publicInstance() {
        return new TestBean("publicInstance");
    }

    // use of a custom qualifier and autowiring of method parameters
    @Bean
    protected TestBean protectedInstance(
            @Qualifier("public") TestBean spouse,
            @Value("#{privateInstance.age}") String country) {
        TestBean tb = new TestBean("protectedInstance", 1);
        tb.setSpouse(spouse);
        tb.setCountry(country);
        return tb;
    }

    @Bean
    private TestBean privateInstance() {
        return new TestBean("privateInstance", i++);
    }

    @Bean
    @RequestScope
    public TestBean requestScopedInstance() {
        return new TestBean("requestScopedInstance", 3);
    }
}
```

The example autowires the `String` method parameter `country` to the value of the `age` property on another bean named `privateInstance`. A Spring Expression Language element defines the value of the property through the notation `#{ <expression> }`. For `@Value` annotations, an expression resolver is preconfigured to look for bean names when resolving expression text.

As of Spring Framework 4.3, you may also declare a factory method parameter of type `InjectionPoint` (or its more specific subclass: `DependencyDescriptor`) to access the requesting injection point that triggers the creation of the current bean. Note that this applies only to the actual creation of bean instances, not to the injection of existing instances. As a consequence, this feature makes most sense for beans of prototype scope. For other scopes, the factory method only ever sees the injection point that triggered the creation of a new bean instance in the given scope (for example, the dependency that triggered the creation of a lazy singleton bean). You can use the provided injection point metadata with semantic care in such scenarios. The following example shows how to do use `InjectionPoint`:

```
@Component
public class FactoryMethodComponent {

    @Bean @Scope("prototype")
    public TestBean prototypeInstance(InjectionPoint injectionPoint) {
        return new TestBean("prototypeInstance for " + injectionPoint.getMember()
);
    }
}
```

The `@Bean` methods in a regular Spring component are processed differently than their counterparts inside a Spring `@Configuration` class. The difference is that `@Component` classes are not enhanced with CGLIB to intercept the invocation of methods and fields. CGLIB proxying is the means by which invoking methods or fields within `@Bean` methods in `@Configuration` classes creates bean metadata references to collaborating objects. Such methods are not invoked with normal Java semantics but rather go through the container in order to provide the usual lifecycle management and proxying of Spring beans, even when referring to other beans through programmatic calls to `@Bean` methods. In contrast, invoking a method or field in a `@Bean` method within a plain `@Component` class has standard Java semantics, with no special CGLIB processing or other constraints applying.

NOTE

You may declare `@Bean` methods as `static`, allowing for them to be called without creating their containing configuration class as an instance. This makes particular sense when defining post-processor beans (for example, of type `BeanFactoryPostProcessor` or `BeanPostProcessor`), since such beans get initialized early in the container lifecycle and should avoid triggering other parts of the configuration at that point.

Calls to static `@Bean` methods never get intercepted by the container, not even within `@Configuration` classes (as described earlier in this section), due to technical limitations: CGLIB subclassing can override only non-static methods. As a consequence, a direct call to another `@Bean` method has standard Java semantics, resulting in an independent instance being returned straight from the factory method itself.

The Java language visibility of `@Bean` methods does not have an immediate impact on the resulting bean definition in Spring's container. You can freely declare your factory methods as you see fit in non-`@Configuration` classes and also for static methods anywhere. However, regular `@Bean` methods in `@Configuration` classes need to be overridable — that is, they must not be declared as `private` or `final`.

`@Bean` methods are also discovered on base classes of a given component or configuration class, as well as on Java 8 default methods declared in interfaces implemented by the component or configuration class. This allows for a lot of flexibility in composing complex configuration arrangements, with even multiple inheritance being possible through Java 8 default methods as of Spring 4.2.

Finally, a single class may hold multiple `@Bean` methods for the same bean, as an arrangement of multiple factory methods to use depending on available dependencies at runtime. This is the same algorithm as for choosing the "greediest" constructor or factory method in other configuration scenarios: The variant with the largest number of satisfiable dependencies is picked at construction time, analogous to how the container selects between multiple `@Autowired` constructors.

Naming Autodetected Components

When a component is autodetected as part of the scanning process, its bean name is generated by the `BeanNameGenerator` strategy known to that scanner. By default, any Spring stereotype annotation (`@Component`, `@Repository`, `@Service`, and `@Controller`) that contains a name `value` thereby provides that name to the corresponding bean definition.

If such an annotation contains no name `value` or for any other detected component (such as those discovered by custom filters), the default bean name generator returns the uncapitalized non-qualified class name. For example, if the following component classes were detected, the names would be `myMovieLister` and `movieFinderImpl`:

```java
@Service("myMovieLister")
public class SimpleMovieLister {
    // ...
}
```

```java
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

NOTE

If you do not want to rely on the default bean-naming strategy, you can provide a custom bean-naming strategy. First, implement the {api-spring-framework}/beans/factory/support/BeanNameGenerator.html[`BeanNameGenerator`] interface, and be sure to include a default no-arg constructor. Then, provide the fully qualified class name when configuring the scanner, as the following example annotation and bean definition show:

```java
@Configuration
@ComponentScan(basePackages = "org.example", nameGenerator = MyNameGenerator.
class)
public class AppConfig {
    ...
}
```

```xml
<beans>
    <context:component-scan base-package="org.example"
        name-generator="org.example.MyNameGenerator" />
</beans>
```

As a general rule, consider specifying the name with the annotation whenever other components may be making explicit references to it. On the other hand, the auto-generated names are adequate whenever the container is responsible for wiring.

Providing a Scope for Autodetected Components

As with Spring-managed components in general, the default and most common scope for autodetected components is `singleton`. However, sometimes you need a different scope that can be specified by the `@Scope` annotation. You can provide the name of the scope within the annotation, as the following example shows:

```java
@Scope("prototype")
@Repository
public class MovieFinderImpl implements MovieFinder {
    // ...
}
```

NOTE `@Scope` annotations are only introspected on the concrete bean class (for annotated components) or the factory method (for `@Bean` methods). In contrast to XML bean definitions, there is no notion of bean definition inheritance, and inheritance hierarchies at the class level are irrelevant for metadata purposes.

For details on web-specific scopes such as “request” or “session” in a Spring context, see Request, Session, Application, and WebSocket Scopes. As with the pre-built annotations for those scopes, you may also compose your own scoping annotations by using Spring’s meta-annotation approach: for example, a custom annotation meta-annotated with `@Scope("prototype")`, possibly also declaring a custom scoped-proxy mode.

NOTE To provide a custom strategy for scope resolution rather than relying on the annotation-based approach, you can implement the {api-spring-framework}/context/annotation/ScopeMetadataResolver.html[`ScopeMetadataResolver`] interface. Be sure to include a default no-arg constructor. Then you can provide the fully qualified class name when configuring the scanner, as the following example of both an annotation and a bean definition shows:

```java
@Configuration
@ComponentScan(basePackages = "org.example", scopeResolver = MyScopeResolver.
class)
public class AppConfig {
    ...
}
```

```xml
<beans>
    <context:component-scan base-package="org.example" scope-resolver=
"org.example.MyScopeResolver"/>
</beans>
```

When using certain non-singleton scopes, it may be necessary to generate proxies for the scoped objects. The reasoning is described in Scoped Beans as Dependencies. For this purpose, a scoped-proxy attribute is available on the component-scan element. The three possible values are: `no`, `interfaces`, and `targetClass`. For example, the following configuration results in standard JDK dynamic proxies:

```java
@Configuration
@ComponentScan(basePackages = "org.example", scopedProxy = ScopedProxyMode
.INTERFACES)
public class AppConfig {
    ...
}
```

```xml
<beans>
    <context:component-scan base-package="org.example" scoped-proxy="interfaces"/>
</beans>
```

Providing Qualifier Metadata with Annotations

The `@Qualifier` annotation is discussed in Fine-tuning Annotation-based Autowiring with Qualifiers. The examples in that section demonstrate the use of the `@Qualifier` annotation and custom qualifier annotations to provide fine-grained control when you resolve autowire candidates. Because those examples were based on XML bean definitions, the qualifier metadata was provided on the candidate bean definitions by using the `qualifier` or `meta` child elements of the `bean` element in the XML. When relying upon classpath scanning for auto-detection of components, you can provide the qualifier metadata with type-level annotations on the candidate class. The following three examples demonstrate this technique:

```java
@Component
@Qualifier("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
```

```java
@Component
@Genre("Action")
public class ActionMovieCatalog implements MovieCatalog {
    // ...
}
```

```java
@Component
@Offline
public class CachingMovieCatalog implements MovieCatalog {
    // ...
}
```

NOTE	As with most annotation-based alternatives, keep in mind that the annotation metadata is bound to the class definition itself, while the use of XML allows for multiple beans of the same type to provide variations in their qualifier metadata, because that metadata is provided per-instance rather than per-class.

Generating an Index of Candidate Components

While classpath scanning is very fast, it is possible to improve the startup performance of large applications by creating a static list of candidates at compilation time. In this mode, all modules that are target of component scan must use this mechanism.

NOTE	Your existing `@ComponentScan` or `<context:component-scan` directives must stay as is to request the context to scan candidates in certain packages. When the `ApplicationContext` detects such an index, it automatically uses it rather than scanning the classpath.

To generate the index, add an additional dependency to each module that contains components that are targets for component scan directives. The following example shows how to do so with Maven:

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context-indexer</artifactId>
        <version>{spring-version}</version>
        <optional>true</optional>
    </dependency>
</dependencies>
```

With Gradle 4.5 and earlier, the dependency should be declared in the `compileOnly` configuration, as shown in the following example:

```
dependencies {
    compileOnly "org.springframework:spring-context-indexer:{spring-version}"
}
```

With Gradle 4.6 and later, the dependency should be declared in the `annotationProcessor` configuration, as shown in the following example:

```
dependencies {
    annotationProcessor "org.springframework:spring-context-indexer:{spring-
version}"
}
```

That process generates a `META-INF/spring.components` file that is included in the jar file.

NOTE When working with this mode in your IDE, the `spring-context-indexer` must be registered as an annotation processor to make sure the index is up-to-date when candidate components are updated.

TIP The index is enabled automatically when a `META-INF/spring.components` is found on the classpath. If an index is partially available for some libraries (or use cases) but could not be built for the whole application, you can fallback to a regular classpath arrangement (as though no index was present at all) by setting `spring.index.ignore` to `true`, either as a system property or in a `spring.properties` file at the root of the classpath.

Using JSR 330 Standard Annotations

Starting with Spring 3.0, Spring offers support for JSR-330 standard annotations (Dependency Injection). Those annotations are scanned in the same way as the Spring annotations. To use them, you need to have the relevant jars in your classpath.

NOTE If you use Maven, the `javax.inject` artifact is available in the standard Maven repository (https://repo1.maven.org/maven2/javax/inject/javax.inject/1/). You can add the following dependency to your file pom.xml:

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

Dependency Injection with `@Inject` and `@Named`

Instead of `@Autowired`, you can use `@javax.inject.Inject` as follows:

```java
import javax.inject.Inject;

public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    public void listMovies() {
        this.movieFinder.findMovies(...);
        ...
    }
}
```

As with `@Autowired`, you can use `@Inject` at the field level, method level and constructor-argument level. Furthermore, you may declare your injection point as a `Provider`, allowing for on-demand access to beans of shorter scopes or lazy access to other beans through a `Provider.get()` call. The following example offers a variant of the preceding example:

```java
import javax.inject.Inject;
import javax.inject.Provider;

public class SimpleMovieLister {

    private Provider<MovieFinder> movieFinder;

    @Inject
    public void setMovieFinder(Provider<MovieFinder> movieFinder) {
        this.movieFinder = movieFinder;
    }

    public void listMovies() {
        this.movieFinder.get().findMovies(...);
        ...
    }
}
```

If you would like to use a qualified name for the dependency that should be injected, you should use the `@Named` annotation, as the following example shows:

```java
import javax.inject.Inject;
import javax.inject.Named;

public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(@Named("main") MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

As with `@Autowired`, `@Inject` can also be used with `java.util.Optional` or `@Nullable`. This is even more applicable here, since `@Inject` does not have a `required` attribute. The following pair of examples show how to use `@Inject` and `@Nullable`:

```java
public class SimpleMovieLister {

    @Inject
    public void setMovieFinder(Optional<MovieFinder> movieFinder) {
        ...
    }
}
```

```java
public class SimpleMovieLister {

    @Inject
    public void setMovieFinder(@Nullable MovieFinder movieFinder) {
        ...
    }
}
```

`@Named` and `@ManagedBean`: Standard Equivalents to the `@Component` Annotation

Instead of `@Component`, you can use `@javax.inject.Named` or `javax.annotation.ManagedBean`, as the following example shows:

```java
import javax.inject.Inject;
import javax.inject.Named;

@Named("movieListener")  // @ManagedBean("movieListener") could be used as well
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

It is very common to use `@Component` without specifying a name for the component. `@Named` can be used in a similar fashion, as the following example shows:

```java
import javax.inject.Inject;
import javax.inject.Named;

@Named
public class SimpleMovieLister {

    private MovieFinder movieFinder;

    @Inject
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // ...
}
```

When you use `@Named` or `@ManagedBean`, you can use component scanning in the exact same way as when you use Spring annotations, as the following example shows:

```java
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    ...
}
```

NOTE In contrast to `@Component`, the JSR-330 `@Named` and the JSR-250 `ManagedBean` annotations are not composable. You should use Spring's stereotype model for building custom component annotations.

Limitations of JSR-330 Standard Annotations

When you work with standard annotations, you should know that some significant features are not available, as the following table shows:

Table 6. Spring component model elements versus JSR-330 variants

Spring	javax.inject.*	javax.inject restrictions / comments
@Autowired	@Inject	`@Inject` has no 'required' attribute. Can be used with Java 8's `Optional` instead.
@Component	@Named / @ManagedBean	JSR-330 does not provide a composable model, only a way to identify named components.
@Scope("singleton")	@Singleton	The JSR-330 default scope is like Spring's `prototype`. However, in order to keep it consistent with Spring's general defaults, a JSR-330 bean declared in the Spring container is a `singleton` by default. In order to use a scope other than `singleton`, you should use Spring's `@Scope` annotation. `javax.inject` also provides a @Scope annotation. Nevertheless, this one is only intended to be used for creating your own annotations.

Spring	javax.inject.*	javax.inject restrictions / comments
@Qualifier	@Qualifier / @Named	`javax.inject.Qualifier` is just a meta-annotation for building custom qualifiers. Concrete `String` qualifiers (like Spring's `@Qualifier` with a value) can be associated through `javax.inject.Named`.
@Value	-	no equivalent
@Required	-	no equivalent
@Lazy	-	no equivalent
ObjectFactory	Provider	`javax.inject.Provider` is a direct alternative to Spring's `ObjectFactory`, only with a shorter `get()` method name. It can also be used in combination with Spring's `@Autowired` or with non-annotated constructors and setter methods.

Java-based Container Configuration

This section covers how to use annotations in your Java code to configure the Spring container. It includes the following topics:

- Basic Concepts: `@Bean` and `@Configuration`
- Instantiating the Spring Container by Using `AnnotationConfigApplicationContext`
- Using the `@Bean` Annotation
- Using the `@Configuration` annotation
- Composing Java-based Configurations
- Bean Definition Profiles
- `PropertySource` Abstraction
- Using `@PropertySource`
- Placeholder Resolution in Statements

Basic Concepts: `@Bean` and `@Configuration`

The central artifacts in Spring's new Java-configuration support are `@Configuration`-annotated classes and `@Bean`-annotated methods.

The `@Bean` annotation is used to indicate that a method instantiates, configures, and initializes a new object to be managed by the Spring IoC container. For those familiar with Spring’s `<beans/>` XML configuration, the `@Bean` annotation plays the same role as the `<bean/>` element. You can use `@Bean` -annotated methods with any Spring `@Component`. However, they are most often used with `@Configuration` beans.

Annotating a class with `@Configuration` indicates that its primary purpose is as a source of bean definitions. Furthermore, `@Configuration` classes let inter-bean dependencies be defined by calling other `@Bean` methods in the same class. The simplest possible `@Configuration` class reads as follows:

```
@Configuration
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

The preceding `AppConfig` class is equivalent to the following Spring `<beans/>` XML:

```
<beans>
    <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

Full @Configuration vs “lite” @Bean mode?

When `@Bean` methods are declared within classes that are not annotated with `@Configuration`, they are referred to as being processed in a “lite” mode. Bean methods declared in a `@Component` or even in a plain old class are considered to be “lite”, with a different primary purpose of the containing class and a `@Bean` method being a sort of bonus there. For example, service components may expose management views to the container through an additional `@Bean` method on each applicable component class. In such scenarios, `@Bean` methods are a general-purpose factory method mechanism.

Unlike full `@Configuration`, lite `@Bean` methods cannot declare inter-bean dependencies. Instead, they operate on their containing component’s internal state and, optionally, on arguments that they may declare. Such a `@Bean` method should therefore not invoke other `@Bean` methods. Each such method is literally only a factory method for a particular bean reference, without any special runtime semantics. The positive side-effect here is that no CGLIB subclassing has to be applied at runtime, so there are no limitations in terms of class design (that is, the containing class may be `final` and so forth).

In common scenarios, `@Bean` methods are to be declared within `@Configuration` classes, ensuring that “full” mode is always used and that cross-method references therefore get redirected to the container’s lifecycle management. This prevents the same `@Bean` method from accidentally being invoked through a regular Java call, which helps to reduce subtle bugs that can be hard to track down when operating in “lite” mode.

The `@Bean` and `@Configuration` annotations are discussed in depth in the following sections. First, however, we cover the various ways of creating a spring container using by Java-based configuration.

Instantiating the Spring Container by Using `AnnotationConfigApplicationContext`

The following sections document Spring’s `AnnotationConfigApplicationContext`, introduced in Spring 3.0. This versatile `ApplicationContext` implementation is capable of accepting not only `@Configuration` classes as input but also plain `@Component` classes and classes annotated with JSR-330 metadata.

When `@Configuration` classes are provided as input, the `@Configuration` class itself is registered as a bean definition and all declared `@Bean` methods within the class are also registered as bean definitions.

When `@Component` and JSR-330 classes are provided, they are registered as bean definitions, and it is assumed that DI metadata such as `@Autowired` or `@Inject` are used within those classes where necessary.

Simple Construction

In much the same way that Spring XML files are used as input when instantiating a

ClassPathXmlApplicationContext, you can use @Configuration classes as input when instantiating an AnnotationConfigApplicationContext. This allows for completely XML-free usage of the Spring container, as the following example shows:

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig
.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

As mentioned earlier, AnnotationConfigApplicationContext is not limited to working only with @Configuration classes. Any @Component or JSR-330 annotated class may be supplied as input to the constructor, as the following example shows:

```
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(MyServiceImpl
.class, Dependency1.class, Dependency2.class);
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

The preceding example assumes that MyServiceImpl, Dependency1, and Dependency2 use Spring dependency injection annotations such as @Autowired.

Building the Container Programmatically by Using register(Class<?>…)

You can instantiate an AnnotationConfigApplicationContext by using a no-arg constructor and then configure it by using the register() method. This approach is particularly useful when programmatically building an AnnotationConfigApplicationContext. The following example shows how to do so:

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
    ctx.register(AppConfig.class, OtherConfig.class);
    ctx.register(AdditionalConfig.class);
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
    myService.doStuff();
}
```

Enabling Component Scanning with `scan(String…)`

To enable component scanning, you can annotate your `@Configuration` class as follows:

```
@Configuration
@ComponentScan(basePackages = "com.acme") ①
public class AppConfig  {
    ...
}
```

① This annotation enables component scanning.

TIP

Experienced Spring users may be familiar with the XML declaration equivalent from Spring's `context:` namespace, shown in the following example:

```
<beans>
    <context:component-scan base-package="com.acme"/>
</beans>
```

In the preceding example, the `com.acme` package is scanned to look for any `@Component`-annotated classes, and those classes are registered as Spring bean definitions within the container. `AnnotationConfigApplicationContext` exposes the `scan(String…)` method to allow for the same component-scanning functionality, as the following example shows:

```
public static void main(String[] args) {
    AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
    ctx.scan("com.acme");
    ctx.refresh();
    MyService myService = ctx.getBean(MyService.class);
}
```

NOTE

Remember that `@Configuration` classes are meta-annotated with `@Component`, so they are candidates for component-scanning. In the preceding example, assuming that `AppConfig` is declared within the `com.acme` package (or any package underneath), it is picked up during the call to `scan()`. Upon `refresh()`, all its `@Bean` methods are processed and registered as bean definitions within the container.

Support for Web Applications with `AnnotationConfigWebApplicationContext`

A `WebApplicationContext` variant of `AnnotationConfigApplicationContext` is available with `AnnotationConfigWebApplicationContext`. You can use this implementation when configuring the Spring `ContextLoaderListener` servlet listener, Spring MVC `DispatcherServlet`, and so forth. The following `web.xml` snippet configures a typical Spring MVC web application (note the use of the `contextClass` context-param and init-param):

```
<web-app>
    <!-- Configure ContextLoaderListener to use
AnnotationConfigWebApplicationContext
        instead of the default XmlWebApplicationContext -->
    <context-param>
        <param-name>contextClass</param-name>
        <param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext
        </param-value>
    </context-param>

    <!-- Configuration locations must consist of one or more comma- or space-
delimited
        fully-qualified @Configuration classes. Fully-qualified packages may also
be
        specified for component-scanning -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>com.acme.AppConfig</param-value>
    </context-param>

    <!-- Bootstrap the root application context as usual using
ContextLoaderListener -->
    <listener>
        <listener-class>
org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- Declare a Spring MVC DispatcherServlet as usual -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <!-- Configure DispatcherServlet to use
AnnotationConfigWebApplicationContext
            instead of the default XmlWebApplicationContext -->
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>

org.springframework.web.context.support.AnnotationConfigWebApplicationContext
            </param-value>
        </init-param>
        <!-- Again, config locations must consist of one or more comma- or space-
delimited
            and fully-qualified @Configuration classes -->
        <init-param>
            <param-name>contextConfigLocation</param-name>
```

```xml
            <param-value>com.acme.web.MvcConfig</param-value>
        </init-param>
    </servlet>

    <!-- map all requests for /app/* to the dispatcher servlet -->
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/app/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Using the `@Bean` Annotation

`@Bean` is a method-level annotation and a direct analog of the XML `<bean/>` element. The annotation supports some of the attributes offered by `<bean/>`, such as: * init-method * destroy-method * autowiring * `name`.

You can use the `@Bean` annotation in a `@Configuration`-annotated or in a `@Component`-annotated class.

Declaring a Bean

To declare a bean, you can annotate a method with the `@Bean` annotation. You use this method to register a bean definition within an `ApplicationContext` of the type specified as the method's return value. By default, the bean name is the same as the method name. The following example shows a `@Bean` method declaration:

```java
@Configuration
public class AppConfig {

    @Bean
    public TransferServiceImpl transferService() {
        return new TransferServiceImpl();
    }
}
```

The preceding configuration is exactly equivalent to the following Spring XML:

```xml
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

Both declarations make a bean named `transferService` available in the `ApplicationContext`, bound to an object instance of type `TransferServiceImpl`, as the following text image shows:

```
transferService -> com.acme.TransferServiceImpl
```

You can also declare your `@Bean` method with an interface (or base class) return type, as the following example shows:

```
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
}
```

However, this limits the visibility for advance type prediction to the specified interface type (`TransferService`). Then, with the full type (`TransferServiceImpl`) known to the container only once, the affected singleton bean has been instantiated. Non-lazy singleton beans get instantiated according to their declaration order, so you may see different type matching results depending on when another component tries to match by a non-declared type (such as `@Autowired TransferServiceImpl`, which resolves only once the `transferService` bean has been instantiated).

TIP If you consistently refer to your types by a declared service interface, your `@Bean` return types may safely join that design decision. However, for components that implement several interfaces or for components potentially referred to by their implementation type, it is safer to declare the most specific return type possible (at least as specific as required by the injection points that refer to your bean).

Bean Dependencies

A `@Bean`-annotated method can have an arbitrary number of parameters that describe the dependencies required to build that bean. For instance, if our `TransferService` requires an `AccountRepository`, we can materialize that dependency with a method parameter, as the following example shows:

```java
@Configuration
public class AppConfig {

    @Bean
    public TransferService transferService(AccountRepository accountRepository) {
        return new TransferServiceImpl(accountRepository);
    }
}
```

The resolution mechanism is pretty much identical to constructor-based dependency injection. See the relevant section for more details.

Receiving Lifecycle Callbacks

Any classes defined with the `@Bean` annotation support the regular lifecycle callbacks and can use the `@PostConstruct` and `@PreDestroy` annotations from JSR-250. See JSR-250 annotations for further details.

The regular Spring lifecycle callbacks are fully supported as well. If a bean implements `InitializingBean`, `DisposableBean`, or `Lifecycle`, their respective methods are called by the container.

The standard set of `*Aware` interfaces (such as BeanFactoryAware, BeanNameAware, MessageSourceAware, ApplicationContextAware, and so on) are also fully supported.

The `@Bean` annotation supports specifying arbitrary initialization and destruction callback methods, much like Spring XML's `init-method` and `destroy-method` attributes on the `bean` element, as the following example shows:

```java
public class BeanOne {

    public void init() {
        // initialization logic
    }
}

public class BeanTwo {

    public void cleanup() {
        // destruction logic
    }
}

@Configuration
public class AppConfig {

    @Bean(initMethod = "init")
    public BeanOne beanOne() {
        return new BeanOne();
    }

    @Bean(destroyMethod = "cleanup")
    public BeanTwo beanTwo() {
        return new BeanTwo();
    }
}
```

NOTE

By default, beans defined with Java configuration that have a public `close` or `shutdown` method are automatically enlisted with a destruction callback. If you have a public `close` or `shutdown` method and you do not wish for it to be called when the container shuts down, you can add `@Bean(destroyMethod="")` to your bean definition to disable the default `(inferred)` mode.

You may want to do that by default for a resource that you acquire with JNDI, as its lifecycle is managed outside the application. In particular, make sure to always do it for a `DataSource`, as it is known to be problematic on Java EE application servers.

The following example shows how to prevent an automatic destruction callback for a `DataSource`:

```
@Bean(destroyMethod="")
public DataSource dataSource() throws NamingException {
    return (DataSource) jndiTemplate.lookup("MyDS");
}
```

Also, with `@Bean` methods, you typically use programmatic JNDI lookups, either by using Spring's `JndiTemplate` or `JndiLocatorDelegate` helpers or straight JNDI `InitialContext` usage but not the `JndiObjectFactoryBean` variant (which would force you to declare the return type as the `FactoryBean` type instead of the actual target type, making it harder to use for cross-reference calls in other `@Bean` methods that intend to refer to the provided resource here).

In the case of `BeanOne` from the example above the preceding note, it would be equally valid to call the `init()` method directly during construction, as the following example shows:

```
@Configuration
public class AppConfig {

    @Bean
    public BeanOne beanOne() {
        BeanOne beanOne = new BeanOne();
        beanOne.init();
        return beanOne;
    }

    // ...
}
```

TIP

When you work directly in Java, you can do anything you like with your objects and do not always need to rely on the container lifecycle.

Specifying Bean Scope

Spring includes the `@Scope` annotation so that you can specify the scope of a bean.

Using the `@Scope` Annotation

You can specify that your beans defined with the `@Bean` annotation should have a specific scope. You can use any of the standard scopes specified in the Bean Scopes section.

The default scope is `singleton`, but you can override this with the `@Scope` annotation, as the following example shows:

```
@Configuration
public class MyConfiguration {

    @Bean
    @Scope("prototype")
    public Encryptor encryptor() {
        // ...
    }
}
```

`@Scope` and `scoped-proxy`

Spring offers a convenient way of working with scoped dependencies through scoped proxies. The easiest way to create such a proxy when using the XML configuration is the `<aop:scoped-proxy/>` element. Configuring your beans in Java with a `@Scope` annotation offers equivalent support with the `proxyMode` attribute. The default is no proxy (`ScopedProxyMode.NO`), but you can specify `ScopedProxyMode.TARGET_CLASS` or `ScopedProxyMode.INTERFACES`.

If you port the scoped proxy example from the XML reference documentation (see scoped proxies) to our `@Bean` using Java, it resembles the following:

```java
// an HTTP Session-scoped bean exposed as a proxy
@Bean
@SessionScope
public UserPreferences userPreferences() {
    return new UserPreferences();
}

@Bean
public Service userService() {
    UserService service = new SimpleUserService();
    // a reference to the proxied userPreferences bean
    service.setUserPreferences(userPreferences());
    return service;
}
```

Customizing Bean Naming

By default, configuration classes use a `@Bean` method's name as the name of the resulting bean. This functionality can be overridden, however, with the `name` attribute, as the following example shows:

```java
@Configuration
public class AppConfig {

    @Bean(name = "myThing")
    public Thing thing() {
        return new Thing();
    }
}
```

Bean Aliasing

As discussed in Naming Beans, it is sometimes desirable to give a single bean multiple names, otherwise known as bean aliasing. The `name` attribute of the `@Bean` annotation accepts a String array for this purpose. The following example shows how to set a number of aliases for a bean:

```java
@Configuration
public class AppConfig {

    @Bean({"dataSource", "subsystemA-dataSource", "subsystemB-dataSource"})
    public DataSource dataSource() {
        // instantiate, configure and return DataSource bean...
    }
}
```

Bean Description

Sometimes, it is helpful to provide a more detailed textual description of a bean. This can be particularly useful when beans are exposed (perhaps through JMX) for monitoring purposes.

To add a description to a `@Bean`, you can use the {api-spring-framework}/context/annotation/Description.html[`@Description`] annotation, as the following example shows:

```java
@Configuration
public class AppConfig {

    @Bean
    @Description("Provides a basic example of a bean")
    public Thing thing() {
        return new Thing();
    }
}
```

Using the `@Configuration` annotation

`@Configuration` is a class-level annotation indicating that an object is a source of bean definitions. `@Configuration` classes declare beans through public `@Bean` annotated methods. Calls to `@Bean` methods on `@Configuration` classes can also be used to define inter-bean dependencies. See Basic Concepts: `@Bean` and `@Configuration` for a general introduction.

Injecting Inter-bean Dependencies

When beans have dependencies on one another, expressing that dependency is as simple as having one bean method call another, as the following example shows:

```java
@Configuration
public class AppConfig {

    @Bean
    public BeanOne beanOne() {
        return new BeanOne(beanTwo());
    }

    @Bean
    public BeanTwo beanTwo() {
        return new BeanTwo();
    }
}
```

In the preceding example, `beanOne` receives a reference to `beanTwo` through constructor injection.

NOTE	This method of declaring inter-bean dependencies works only when the `@Bean` method is declared within a `@Configuration` class. You cannot declare inter-bean dependencies by using plain `@Component` classes.

Lookup Method Injection

As noted earlier, lookup method injection is an advanced feature that you should use rarely. It is useful in cases where a singleton-scoped bean has a dependency on a prototype-scoped bean. Using Java for this type of configuration provides a natural means for implementing this pattern. The following example shows how to use lookup method injection:

```java
public abstract class CommandManager {
    public Object process(Object commandState) {
        // grab a new instance of the appropriate Command interface
        Command command = createCommand();
        // set the state on the (hopefully brand new) Command instance
        command.setState(commandState);
        return command.execute();
    }

    // okay... but where is the implementation of this method?
    protected abstract Command createCommand();
}
```

By using Java configuration, you can create a subclass of `CommandManager` where the abstract `createCommand()` method is overridden in such a way that it looks up a new (prototype) command object. The following example shows how to do so:

```java
@Bean
@Scope("prototype")
public AsyncCommand asyncCommand() {
    AsyncCommand command = new AsyncCommand();
    // inject dependencies here as required
    return command;
}

@Bean
public CommandManager commandManager() {
    // return new anonymous implementation of CommandManager with createCommand()
    // overridden to return a new prototype Command object
    return new CommandManager() {
        protected Command createCommand() {
            return asyncCommand();
        }
    }
}
```

Further Information About How Java-based Configuration Works Internally

Consider the following example, which shows a `@Bean` annotated method being called twice:

```java
@Configuration
public class AppConfig {

    @Bean
    public ClientService clientService1() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientService clientService2() {
        ClientServiceImpl clientService = new ClientServiceImpl();
        clientService.setClientDao(clientDao());
        return clientService;
    }

    @Bean
    public ClientDao clientDao() {
        return new ClientDaoImpl();
    }
}
```

`clientDao()` has been called once in `clientService1()` and once in `clientService2()`. Since this method creates a new instance of `ClientDaoImpl` and returns it, you would normally expect to have two instances (one for each service). That definitely would be problematic: In Spring, instantiated beans have a `singleton` scope by default. This is where the magic comes in: All `@Configuration` classes are subclassed at startup-time with `CGLIB`. In the subclass, the child method checks the container first for any cached (scoped) beans before it calls the parent method and creates a new instance.

NOTE The behavior could be different according to the scope of your bean. We are talking about singletons here.

NOTE As of Spring 3.2, it is no longer necessary to add CGLIB to your classpath because CGLIB classes have been repackaged under `org.springframework.cglib` and included directly within the spring-core JAR.

TIP There are a few restrictions due to the fact that CGLIB dynamically adds features at startup-time. In particular, configuration classes must not be final. However, as of 4.3, any constructors are allowed on configuration classes, including the use of `@Autowired` or a single non-default constructor declaration for default injection.

If you prefer to avoid any CGLIB-imposed limitations, consider declaring your `@Bean` methods on non-`@Configuration` classes (for example, on plain `@Component` classes instead). Cross-method calls between `@Bean` methods are not then intercepted, so you have to exclusively rely on dependency injection at the constructor or method level there.

Composing Java-based Configurations

Spring's Java-based configuration feature lets you compose annotations, which can reduce the complexity of your configuration.

Using the `@Import` Annotation

Much as the `<import/>` element is used within Spring XML files to aid in modularizing configurations, the `@Import` annotation allows for loading `@Bean` definitions from another configuration class, as the following example shows:

```java
@Configuration
public class ConfigA {

    @Bean
    public A a() {
        return new A();
    }
}

@Configuration
@Import(ConfigA.class)
public class ConfigB {

    @Bean
    public B b() {
        return new B();
    }
}
```

Now, rather than needing to specify both `ConfigA.class` and `ConfigB.class` when instantiating the context, only `ConfigB` needs to be supplied explicitly, as the following example shows:

```java
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(ConfigB.class
);

    // now both beans A and B will be available...
    A a = ctx.getBean(A.class);
    B b = ctx.getBean(B.class);
}
```

This approach simplifies container instantiation, as only one class needs to be dealt with, rather than requiring you to remember a potentially large number of `@Configuration` classes during construction.

TIP	As of Spring Framework 4.2, `@Import` also supports references regular component classes, analogous to the `AnnotationConfigApplicationContext.register` method. This is particularly useful if you want to avoid component scanning, by using a few configuration classes as entry points to explicitly define all your components.

Injecting Dependencies on Imported `@Bean` Definitions

The preceding example works but is simplistic. In most practical scenarios, beans have dependencies on one another across configuration classes. When using XML, this is not an issue, because no compiler is involved, and you can declare `ref="someBean"` and trust Spring to work it out during container initialization. When using `@Configuration` classes, the Java compiler places constraints on the configuration model, in that references to other beans must be valid Java syntax.

Fortunately, solving this problem is simple. As we already discussed, a `@Bean` method can have an arbitrary number of parameters that describe the bean dependencies. Consider the following more real-world scenario with several `@Configuration` classes, each depending on beans declared in the others:

```
@Configuration
public class ServiceConfig {

    @Bean
    public TransferService transferService(AccountRepository accountRepository) {
        return new TransferServiceImpl(accountRepository);
    }
}

@Configuration
public class RepositoryConfig {

    @Bean
    public AccountRepository accountRepository(DataSource dataSource) {
        return new JdbcAccountRepository(dataSource);
    }
}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return new DataSource
    }
}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext
(SystemTestConfig.class);
    // everything wires up across configuration classes...
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

There is another way to achieve the same result. Remember that `@Configuration` classes are ultimately only another bean in the container: This means that they can take advantage of `@Autowired` and `@Value` injection and other features the same as any other bean.

WARNING

Make sure that the dependencies you inject that way are of the simplest kind only. `@Configuration` classes are processed quite early during the initialization of the context, and forcing a dependency to be injected this way may lead to unexpected early initialization. Whenever possible, resort to parameter-based injection, as in the preceding example.

Also, be particularly careful with `BeanPostProcessor` and `BeanFactoryPostProcessor` definitions through `@Bean`. Those should usually be declared as `static @Bean` methods, not triggering the instantiation of their containing configuration class. Otherwise, `@Autowired` and `@Value` do not work on the configuration class itself, since it is being created as a bean instance too early.

The following example shows how one bean can be autowired to another bean:

```java
@Configuration
public class ServiceConfig {

    @Autowired
    private AccountRepository accountRepository;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(accountRepository);
    }
}

@Configuration
public class RepositoryConfig {

    private final DataSource dataSource;

    @Autowired
    public RepositoryConfig(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }
}

@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return new DataSource
    }
}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext
(SystemTestConfig.class);
    // everything wires up across configuration classes...
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

TIP Constructor injection in `@Configuration` classes is only supported as of Spring Framework 4.3. Note also that there is no need to specify `@Autowired` if the target bean defines only one constructor. In the preceding example, `@Autowired` is not necessary on the `RepositoryConfig` constructor.

Fully-qualifying imported beans for ease of navigation

In the preceding scenario, using `@Autowired` works well and provides the desired modularity, but determining exactly where the autowired bean definitions are declared is still somewhat ambiguous. For example, as a developer looking at `ServiceConfig`, how do you know exactly where the `@Autowired AccountRepository` bean is declared? It is not explicit in the code, and this may be just fine. Remember that the Spring Tool Suite provides tooling that can render graphs showing how everything is wired, which may be all you need. Also, your Java IDE can easily find all declarations and uses of the `AccountRepository` type and quickly show you the location of `@Bean` methods that return that type.

In cases where this ambiguity is not acceptable and you wish to have direct navigation from within your IDE from one `@Configuration` class to another, consider autowiring the configuration classes themselves. The following example shows how to do so:

```
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        // navigate 'through' the config class to the @Bean method!
        return new TransferServiceImpl(repositoryConfig.accountRepository());
    }
}
```

In the preceding situation, where `AccountRepository` is defined is completely explicit. However, `ServiceConfig` is now tightly coupled to `RepositoryConfig`. That is the tradeoff. This tight coupling can be somewhat mitigated by using interface-based or abstract class-based `@Configuration` classes. Consider the following example:

```java
@Configuration
public class ServiceConfig {

    @Autowired
    private RepositoryConfig repositoryConfig;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(repositoryConfig.accountRepository());
    }
}

@Configuration
public interface RepositoryConfig {

    @Bean
    AccountRepository accountRepository();
}

@Configuration
public class DefaultRepositoryConfig implements RepositoryConfig {

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(...);
    }
}

@Configuration
@Import({ServiceConfig.class, DefaultRepositoryConfig.class})  // import the
concrete config!
public class SystemTestConfig {

    @Bean
    public DataSource dataSource() {
        // return DataSource
    }

}

public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext
(SystemTestConfig.class);
    TransferService transferService = ctx.getBean(TransferService.class);
    transferService.transfer(100.00, "A123", "C456");
}
```

Now `ServiceConfig` is loosely coupled with respect to the concrete `DefaultRepositoryConfig`, and built-in IDE tooling is still useful: You can easily get a type hierarchy of `RepositoryConfig` implementations. In this way, navigating `@Configuration` classes and their dependencies becomes no different than the usual process of navigating interface-based code.

TIP	If you want to influence the startup creation order of certain beans, consider declaring some of them as `@Lazy` (for creation on first access instead of on startup) or as `@DependsOn` certain other beans (making sure that specific other beans are created before the current bean, beyond what the latter's direct dependencies imply).

Conditionally Include `@Configuration` Classes or `@Bean` Methods

It is often useful to conditionally enable or disable a complete `@Configuration` class or even individual `@Bean` methods, based on some arbitrary system state. One common example of this is to use the `@Profile` annotation to activate beans only when a specific profile has been enabled in the Spring `Environment` (see Bean Definition Profiles for details).

The `@Profile` annotation is actually implemented by using a much more flexible annotation called {api-spring-framework}/context/annotation/Conditional.html[`@Conditional`]. The `@Conditional` annotation indicates specific `org.springframework.context.annotation.Condition` implementations that should be consulted before a `@Bean` is registered.

Implementations of the `Condition` interface provide a `matches(…)` method that returns `true` or `false`. For example, the following listing shows the actual `Condition` implementation used for `@Profile`:

```
@Override
public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
    if (context.getEnvironment() != null) {
        // Read the @Profile annotation attributes
        MultiValueMap<String, Object> attrs = metadata.getAllAnnotationAttributes
(Profile.class.getName());
        if (attrs != null) {
            for (Object value : attrs.get("value")) {
                if (context.getEnvironment().acceptsProfiles(((String[]) value)))
{
                    return true;
                }
            }
            return false;
        }
    }
    return true;
}
```

See the {api-spring-framework}/context/annotation/Conditional.html[`@Conditional`] javadoc for more detail.

Combining Java and XML Configuration

Spring's `@Configuration` class support does not aim to be a 100% complete replacement for Spring XML. Some facilities, such as Spring XML namespaces, remain an ideal way to configure the container. In cases where XML is convenient or necessary, you have a choice: either instantiate the container in an "XML-centric" way by using, for example, `ClassPathXmlApplicationContext`, or instantiate it in a "Java-centric" way by using `AnnotationConfigApplicationContext` and the `@ImportResource` annotation to import XML as needed.

XML-centric Use of `@Configuration` Classes

It may be preferable to bootstrap the Spring container from XML and include `@Configuration` classes in an ad-hoc fashion. For example, in a large existing codebase that uses Spring XML, it is easier to create `@Configuration` classes on an as-needed basis and include them from the existing XML files. Later in this section, we cover the options for using `@Configuration` classes in this kind of "XML-centric" situation.

Declaring `@Configuration` classes as plain Spring `<bean/>` elements

Remember that `@Configuration` classes are ultimately bean definitions in the container. In this series examples, we create a `@Configuration` class named `AppConfig` and include it within `system-test-config.xml` as a `<bean/>` definition. Because `<context:annotation-config/>` is switched on, the container recognizes the `@Configuration` annotation and processes the `@Bean` methods declared in `AppConfig` properly.

The following example shows an ordinary configuration class in Java:

```java
@Configuration
public class AppConfig {

    @Autowired
    private DataSource dataSource;

    @Bean
    public AccountRepository accountRepository() {
        return new JdbcAccountRepository(dataSource);
    }

    @Bean
    public TransferService transferService() {
        return new TransferService(accountRepository());
    }
}
```

The following example shows part of a sample `system-test-config.xml` file:

```xml
<beans>
    <!-- enable processing of annotations such as @Autowired and @Configuration
-->
    <context:annotation-config/>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

    <bean class="com.acme.AppConfig"/>

    <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

The following example shows a possible `jdbc.properties` file:

```
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=
```

```java
public static void main(String[] args) {
    ApplicationContext ctx = new ClassPathXmlApplicationContext(
"classpath:/com/acme/system-test-config.xml");
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}
```

NOTE In `system-test-config.xml` file, the `AppConfig` `<bean/>` does not declare an `id` element. While it would be acceptable to do so, it is unnecessary, given that no other bean ever refers to it, and it is unlikely to be explicitly fetched from the container by name. Similarly, the `DataSource` bean is only ever autowired by type, so an explicit bean `id` is not strictly required.

Using <context:component-scan/> to pick up `@Configuration` classes

Because `@Configuration` is meta-annotated with `@Component`, `@Configuration`-annotated classes are automatically candidates for component scanning. Using the same scenario as describe in the previous example, we can redefine `system-test-config.xml` to take advantage of component-scanning. Note that, in this case, we need not explicitly declare `<context:annotation-config/>`, because `<context:component-scan/>` enables the same functionality.

The following example shows the modified `system-test-config.xml` file:

```xml
<beans>
    <!-- picks up and registers AppConfig as a bean definition -->
    <context:component-scan base-package="com.acme"/>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>

    <bean class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>
</beans>
```

`@Configuration` Class-centric Use of XML with `@ImportResource`

In applications where `@Configuration` classes are the primary mechanism for configuring the container, it is still likely necessary to use at least some XML. In these scenarios, you can use `@ImportResource` and define only as much XML as you need. Doing so achieves a "Java-centric" approach to configuring the container and keeps XML to a bare minimum. The following example (which includes a configuration class, an XML file that defines a bean, a properties file, and the `main` class) shows how to use the `@ImportResource` annotation to achieve "Java-centric" configuration that uses XML as needed:

```java
@Configuration
@ImportResource("classpath:/com/acme/properties-config.xml")
public class AppConfig {

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;

    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(url, username, password);
    }
}
```

```
properties-config.xml
<beans>
    <context:property-placeholder location="classpath:/com/acme/jdbc.properties"/>
</beans>
```

```
jdbc.properties
jdbc.url=jdbc:hsqldb:hsql://localhost/xdb
jdbc.username=sa
jdbc.password=
```

```java
public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig
.class);
    TransferService transferService = ctx.getBean(TransferService.class);
    // ...
}
```

Environment Abstraction

The {api-spring-framework}/core/env/Environment.html[`Environment`] interface is an abstraction integrated in the container that models two key aspects of the application environment: profiles and properties.

A profile is a named, logical group of bean definitions to be registered with the container only if the given profile is active. Beans may be assigned to a profile whether defined in XML or with annotations. The role of the `Environment` object with relation to profiles is in determining which profiles (if any) are currently active, and which profiles (if any) should be active by default.

Properties play an important role in almost all applications and may originate from a variety of sources: properties files, JVM system properties, system environment variables, JNDI, servlet context parameters, ad-hoc `Properties` objects, `Map` objects, and so on. The role of the `Environment` object with relation to properties is to provide the user with a convenient service interface for configuring property sources and resolving properties from them.

Bean Definition Profiles

Bean definition profiles provide a mechanism in the core container that allows for registration of different beans in different environments. The word, “environment,” can mean different things to different users, and this feature can help with many use cases, including:

- Working against an in-memory datasource in development versus looking up that same datasource from JNDI when in QA or production.
- Registering monitoring infrastructure only when deploying an application into a performance environment.
- Registering customized implementations of beans for customer A versus customer B deployments.

Consider the first use case in a practical application that requires a `DataSource`. In a test environment, the configuration might resemble the following:

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("my-schema.sql")
        .addScript("my-test-data.sql")
        .build();
}
```

Now consider how this application can be deployed into a QA or production environment, assuming that the datasource for the application is registered with the production application server’s JNDI directory. Our `dataSource` bean now looks like the following listing:

```
@Bean(destroyMethod="")
public DataSource dataSource() throws Exception {
    Context ctx = new InitialContext();
    return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
}
```

The problem is how to switch between using these two variations based on the current environment. Over time, Spring users have devised a number of ways to get this done, usually relying on a combination of system environment variables and XML `<import/>` statements containing `${placeholder}` tokens that resolve to the correct configuration file path depending on the value of an environment variable. Bean definition profiles is a core container feature that provides a solution to this problem.

If we generalize the use case shown in the preceding example of environment-specific bean definitions, we end up with the need to register certain bean definitions in certain contexts but not in others. You could say that you want to register a certain profile of bean definitions in situation A and a different profile in situation B. We start by updating our configuration to reflect this need.

Using `@Profile`

The {api-spring-framework}/context/annotation/Profile.html[`@Profile`] annotation lets you indicate that a component is eligible for registration when one or more specified profiles are active. Using our preceding example, we can rewrite the `dataSource` configuration as follows:

```java
@Configuration
@Profile("development")
public class StandaloneDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }
}
```

```java
@Configuration
@Profile("production")
public class JndiDataConfig {

    @Bean(destroyMethod="")
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

NOTE As mentioned earlier, with `@Bean` methods, you typically choose to use programmatic JNDI lookups, by using either Spring's `JndiTemplate` /`JndiLocatorDelegate` helpers or the straight JNDI `InitialContext` usage shown earlier but not the `JndiObjectFactoryBean` variant, which would force you to declare the return type as the `FactoryBean` type.

The profile string may contain a simple profile name (for example, `production`) or a profile expression. A profile expression allows for more complicated profile logic to be expressed (for example, `production & us-east`). The following operators are supported in profile expressions:

- `!`: A logical "not" of the profile
- `&`: A logical "and" of the profiles
- `|`: A logical "or" of the profiles

NOTE You cannot mix the `&` and `|` operators without using parentheses. For example, `production & us-east | eu-central` is not a valid expression. It must be expressed as `production & (us-east | eu-central)`.

You can use `@Profile` as a meta-annotation for the purpose of creating a custom composed annotation. The following example defines a custom `@Production` annotation that you can use as a drop-in replacement for `@Profile("production")`:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("production")
public @interface Production {
}
```

TIP If a `@Configuration` class is marked with `@Profile`, all of the `@Bean` methods and `@Import` annotations associated with that class are bypassed unless one or more of the specified profiles are active. If a `@Component` or `@Configuration` class is marked with `@Profile({"p1", "p2"})`, that class is not registered or processed unless profiles 'p1' or 'p2' have been activated. If a given profile is prefixed with the NOT operator (`!`), the annotated element is registered only if the profile is not active. For example, given `@Profile({"p1", "!p2"})`, registration will occur if profile 'p1' is active or if profile 'p2' is not active.

`@Profile` can also be declared at the method level to include only one particular bean of a configuration class (for example, for alternative variants of a particular bean), as the following example shows:

```java
@Configuration
public class AppConfig {

    @Bean("dataSource")
    @Profile("development") ①
    public DataSource standaloneDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }

    @Bean("dataSource")
    @Profile("production") ②
    public DataSource jndiDataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```

① The `standaloneDataSource` method is available only in the `development` profile.

② The `jndiDataSource` method is available only in the `production` profile.

NOTE

With `@Profile` on `@Bean` methods, a special scenario may apply: In the case of overloaded `@Bean` methods of the same Java method name (analogous to constructor overloading), a `@Profile` condition needs to be consistently declared on all overloaded methods. If the conditions are inconsistent, only the condition on the first declaration among the overloaded methods matters. Therefore, `@Profile` can not be used to select an overloaded method with a particular argument signature over another. Resolution between all factory methods for the same bean follows Spring's constructor resolution algorithm at creation time.

If you want to define alternative beans with different profile conditions, use distinct Java method names that point to the same bean name by using the `@Bean` name attribute, as shown in the preceding example. If the argument signatures are all the same (for example, all of the variants have no-arg factory methods), this is the only way to represent such an arrangement in a valid Java class in the first place (since there can only be one method of a particular name and argument signature).

XML Bean Definition Profiles

The XML counterpart is the `profile` attribute of the `<beans>` element. Our preceding sample configuration can be rewritten in two XML files, as follows:

```
<beans profile="development"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="...">

    <jdbc:embedded-database id="dataSource">
        <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
        <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
    </jdbc:embedded-database>
</beans>
```

```
<beans profile="production"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource"/>
</beans>
```

It is also possible to avoid that split and nest `<beans/>` elements within the same file, as the following example shows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="...">

    <!-- other bean definitions -->

    <beans profile="development">
        <jdbc:embedded-database id="dataSource">
            <jdbc:script location="classpath:com/bank/config/sql/schema.sql"/>
            <jdbc:script location="classpath:com/bank/config/sql/test-data.sql"/>
        </jdbc:embedded-database>
    </beans>

    <beans profile="production">
        <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/datasource
"/>
    </beans>
</beans>
```

The `spring-bean.xsd` has been constrained to allow such elements only as the last ones in the file. This should help provide flexibility without incurring clutter in the XML files.

> **NOTE**
>
> The XML counterpart does not support the profile expressions described earlier. It is possible, however, to negate a profile by using the `!` operator. It is also possible to apply a logical “and” by nesting the profiles, as the following example shows:
>
> ```
> <beans xmlns="http://www.springframework.org/schema/beans"
>     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
>     xmlns:jdbc="http://www.springframework.org/schema/jdbc"
>     xmlns:jee="http://www.springframework.org/schema/jee"
>     xsi:schemaLocation="...">
>
>     <!-- other bean definitions -->
>
>     <beans profile="production">
>         <beans profile="us-east">
>             <jee:jndi-lookup id="dataSource" jndi-name=
> "java:comp/env/jdbc/datasource"/>
>         </beans>
>     </beans>
> </beans>
> ```
>
> In the preceding example, the `dataSource` bean is exposed if both the `production` and `us-east` profiles are active.

Activating a Profile

Now that we have updated our configuration, we still need to instruct Spring which profile is active. If we started our sample application right now, we would see a `NoSuchBeanDefinitionException` thrown, because the container could not find the Spring bean named `dataSource`.

Activating a profile can be done in several ways, but the most straightforward is to do it programmatically against the `Environment` API which is available through an `ApplicationContext`. The following example shows how to do so:

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("development");
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);
ctx.refresh();
```

In addition, you can also declaratively activate profiles through the `spring.profiles.active` property, which may be specified through system environment variables, JVM system properties, servlet context parameters in `web.xml`, or even as an entry in JNDI (see `PropertySource` Abstraction). In integration tests, active profiles can be declared by using the `@ActiveProfiles` annotation in the `spring-test` module (see context configuration with environment profiles).

Note that profiles are not an “either-or” proposition. You can activate multiple profiles at once. Programmatically, you can provide multiple profile names to the `setActiveProfiles()` method, which accepts `String…` varargs. The following example activates multiple profiles:

```
ctx.getEnvironment().setActiveProfiles("profile1", "profile2");
```

Declaratively, `spring.profiles.active` may accept a comma-separated list of profile names, as the following example shows:

```
-Dspring.profiles.active="profile1,profile2"
```

Default Profile

The default profile represents the profile that is enabled by default. Consider the following example:

```
@Configuration
@Profile("default")
public class DefaultDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .build();
    }
}
```

If no profile is active, the `dataSource` is created. You can see this as a way to provide a default definition for one or more beans. If any profile is enabled, the default profile does not apply.

You can change the name of the default profile by using `setDefaultProfiles()` on the `Environment` or ,declaratively, by using the `spring.profiles.default` property.

`PropertySource` Abstraction

Spring's `Environment` abstraction provides search operations over a configurable hierarchy of property sources. Consider the following listing:

```
ApplicationContext ctx = new GenericApplicationContext();
Environment env = ctx.getEnvironment();
boolean containsMyProperty = env.containsProperty("my-property");
System.out.println("Does my environment contain the 'my-property' property? " +
containsMyProperty);
```

In the preceding snippet, we see a high-level way of asking Spring whether the `my-property` property is defined for the current environment. To answer this question, the `Environment` object performs a search over a set of {api-spring-framework}/core/env/PropertySource.html[`PropertySource`] objects. A `PropertySource` is a simple abstraction over any source of key-value pairs, and Spring's {api-spring-framework}/core/env/StandardEnvironment.html[`StandardEnvironment`] is configured with two PropertySource objects — one representing the set of JVM system properties (`System.getProperties()`) and one representing the set of system environment variables (`System.getenv()`).

NOTE	These default property sources are present for `StandardEnvironment`, for use in standalone applications. {api-spring-framework}/web/context/support/StandardServletEnvironment.html[`StandardServletEnvironment`] is populated with additional default property sources including servlet config and servlet context parameters. It can optionally enable a {api-spring-framework}/jndi/JndiPropertySource.html[`JndiPropertySource`]. See the javadoc for details.

Concretely, when you use the `StandardEnvironment`, the call to `env.containsProperty("my-property")` returns true if a `my-property` system property or `my-property` environment variable is present at runtime.

TIP

> The search performed is hierarchical. By default, system properties have precedence over environment variables. So, if the `my-property` property happens to be set in both places during a call to `env.getProperty("my-property")`, the system property value “wins” and is returned. Note that property values are not merged but rather completely overridden by a preceding entry.
>
> For a common `StandardServletEnvironment`, the full hierarchy is as follows, with the highest-precedence entries at the top:
>
> 1. ServletConfig parameters (if applicable — for example, in case of a `DispatcherServlet` context)
> 2. ServletContext parameters (web.xml context-param entries)
> 3. JNDI environment variables (`java:comp/env/` entries)
> 4. JVM system properties (`-D` command-line arguments)
> 5. JVM system environment (operating system environment variables)

Most importantly, the entire mechanism is configurable. Perhaps you have a custom source of properties that you want to integrate into this search. To do so, implement and instantiate your own `PropertySource` and add it to the set of `PropertySources` for the current `Environment`. The following example shows how to do so:

```
ConfigurableApplicationContext ctx = new GenericApplicationContext();
MutablePropertySources sources = ctx.getEnvironment().getPropertySources();
sources.addFirst(new MyPropertySource());
```

In the preceding code, `MyPropertySource` has been added with highest precedence in the search. If it contains a `my-property` property, the property is detected and returned, in favor of any `my-property` property in any other `PropertySource`. The {api-spring-framework}/core/env/MutablePropertySources.html[`MutablePropertySources`] API exposes a number of methods that allow for precise manipulation of the set of property sources.

Using `@PropertySource`

The {api-spring-framework}/context/annotation/PropertySource.html[`@PropertySource`] annotation provides a convenient and declarative mechanism for adding a `PropertySource` to Spring’s `Environment`.

Given a file called `app.properties` that contains the key-value pair `testbean.name=myTestBean`, the following `@Configuration` class uses `@PropertySource` in such a way that a call to `testBean.getName()` returns `myTestBean`:

```
@Configuration
@PropertySource("classpath:/com/myco/app.properties")
public class AppConfig {

 @Autowired
 Environment env;

 @Bean
 public TestBean testBean() {
  TestBean testBean = new TestBean();
  testBean.setName(env.getProperty("testbean.name"));
  return testBean;
 }
}
```

Any `${…}` placeholders present in a `@PropertySource` resource location are resolved against the set of property sources already registered against the environment, as the following example shows:

```
@Configuration
@PropertySource("classpath:/com/${my.placeholder:default/path}/app.properties")
public class AppConfig {

 @Autowired
 Environment env;

 @Bean
 public TestBean testBean() {
  TestBean testBean = new TestBean();
  testBean.setName(env.getProperty("testbean.name"));
  return testBean;
 }
}
```

Assuming that `my.placeholder` is present in one of the property sources already registered (for example, system properties or environment variables), the placeholder is resolved to the corresponding value. If not, then `default/path` is used as a default. If no default is specified and a property cannot be resolved, an `IllegalArgumentException` is thrown.

NOTE The `@PropertySource` annotation is repeatable, according to Java 8 conventions. However, all such `@PropertySource` annotations need to be declared at the same level, either directly on the configuration class or as meta-annotations within the same custom annotation. Mixing direct annotations and meta-annotations is not recommended, since direct annotations effectively override meta-annotations.

Placeholder Resolution in Statements

Historically, the value of placeholders in elements could be resolved only against JVM system properties or environment variables. This is no longer the case. Because the `Environment` abstraction is integrated throughout the container, it is easy to route resolution of placeholders through it. This means that you may configure the resolution process in any way you like. You can change the precedence of searching through system properties and environment variables or remove them entirely. You can also add your own property sources to the mix, as appropriate.

Concretely, the following statement works regardless of where the `customer` property is defined, as long as it is available in the `Environment`:

```
<beans>
    <import resource="com/bank/service/${customer}-config.xml"/>
</beans>
```

Registering a `LoadTimeWeaver`

The `LoadTimeWeaver` is used by Spring to dynamically transform classes as they are loaded into the Java virtual machine (JVM).

To enable load-time weaving, you can add the `@EnableLoadTimeWeaving` to one of your `@Configuration` classes, as the following example shows:

```
@Configuration
@EnableLoadTimeWeaving
public class AppConfig {
}
```

Alternatively, for XML configuration, you can use the `context:load-time-weaver` element:

```
<beans>
    <context:load-time-weaver/>
</beans>
```

Once configured for the `ApplicationContext`, any bean within that `ApplicationContext` may implement `LoadTimeWeaverAware`, thereby receiving a reference to the load-time weaver instance. This is particularly useful in combination with Spring's JPA support where load-time weaving may be necessary for JPA class transformation. Consult the {api-spring-framework}/orm/jpa/LocalContainerEntityManagerFactoryBean.html[`LocalContainerEntityManagerF` actoryBean] javadoc for more detail. For more on AspectJ load-time weaving, see [aop-aj-ltw].

Additional Capabilities of the ApplicationContext

As discussed in the chapter introduction, the org.springframework.beans.factory package provides basic functionality for managing and manipulating beans, including in a programmatic way. The org.springframework.context package adds the {api-spring-framework}/context/ApplicationContext.html[ApplicationContext] interface, which extends the BeanFactory interface, in addition to extending other interfaces to provide additional functionality in a more application framework-oriented style. Many people use the ApplicationContext in a completely declarative fashion, not even creating it programmatically, but instead relying on support classes such as ContextLoader to automatically instantiate an ApplicationContext as part of the normal startup process of a Java EE web application.

To enhance BeanFactory functionality in a more framework-oriented style, the context package also provides the following functionality:

- Access to messages in i18n-style, through the MessageSource interface.
- Access to resources, such as URLs and files, through the ResourceLoader interface.
- Event publication, namely to beans that implement the ApplicationListener interface, through the use of the ApplicationEventPublisher interface.
- Loading of multiple (hierarchical) contexts, letting each be focused on one particular layer, such as the web layer of an application, through the HierarchicalBeanFactory interface.

Internationalization using MessageSource

The ApplicationContext interface extends an interface called MessageSource and, therefore, provides internationalization (“i18n”) functionality. Spring also provides the HierarchicalMessageSource interface, which can resolve messages hierarchically. Together, these interfaces provide the foundation upon which Spring effects message resolution. The methods defined on these interfaces include:

- String getMessage(String code, Object[] args, String default, Locale loc): The basic method used to retrieve a message from the MessageSource. When no message is found for the specified locale, the default message is used. Any arguments passed in become replacement values, using the MessageFormat functionality provided by the standard library.
- String getMessage(String code, Object[] args, Locale loc): Essentially the same as the previous method but with one difference: No default message can be specified. If the message cannot be found, a NoSuchMessageException is thrown.
- String getMessage(MessageSourceResolvable resolvable, Locale locale): All properties used in the preceding methods are also wrapped in a class named MessageSourceResolvable, which you can use with this method.

When an ApplicationContext is loaded, it automatically searches for a MessageSource bean defined in the context. The bean must have the name `messageSource`. If such a bean is found, all calls to the preceding methods are delegated to the message source. If no message source is found, the `ApplicationContext` attempts to find a parent containing a bean with the same name. If it does, it uses that bean as the `MessageSource`. If the `ApplicationContext` cannot find any source for messages, an empty `DelegatingMessageSource` is instantiated in order to be able to accept calls to the methods defined above.

Spring provides two `MessageSource` implementations, `ResourceBundleMessageSource` and `StaticMessageSource`. Both implement `HierarchicalMessageSource` in order to do nested messaging. The `StaticMessageSource` is rarely used but provides programmatic ways to add messages to the source. The following example shows `ResourceBundleMessageSource`:

```
<beans>
    <bean id="messageSource"
            class="
org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basenames">
            <list>
                <value>format</value>
                <value>exceptions</value>
                <value>windows</value>
            </list>
        </property>
    </bean>
</beans>
```

The example assumes that you have three resource bundles called `format`, `exceptions` and `windows` defined in your classpath. Any request to resolve a message is handled in the JDK-standard way of resolving messages through `ResourceBundle` objects. For the purposes of the example, assume the contents of two of the above resource bundle files are as follows:

```
# in format.properties
message=Alligators rock!
```

```
# in exceptions.properties
argument.required=The {0} argument is required.
```

The next example shows a program to execute the `MessageSource` functionality. Remember that all `ApplicationContext` implementations are also `MessageSource` implementations and so can be cast to the `MessageSource` interface.

```java
public static void main(String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("message", null, "Default", null);
    System.out.println(message);
}
```

The resulting output from the above program is as follows:

```
Alligators rock!
```

To summarize, the `MessageSource` is defined in a file called `beans.xml`, which exists at the root of your classpath. The `messageSource` bean definition refers to a number of resource bundles through its `basenames` property. The three files that are passed in the list to the `basenames` property exist as files at the root of your classpath and are called `format.properties`, `exceptions.properties`, and `windows.properties`, respectively.

The next example shows arguments passed to the message lookup. These arguments are converted into `String` objects and inserted into placeholders in the lookup message.

```xml
<beans>

    <!-- this MessageSource is being used in a web application -->
    <bean id="messageSource" class=
"org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="exceptions"/>
    </bean>

    <!-- lets inject the above MessageSource into this POJO -->
    <bean id="example" class="com.something.Example">
        <property name="messages" ref="messageSource"/>
    </bean>

</beans>
```

```java
public class Example {

    private MessageSource messages;

    public void setMessages(MessageSource messages) {
        this.messages = messages;
    }

    public void execute() {
        String message = this.messages.getMessage("argument.required",
            new Object [] {"userDao"}, "Required", null);
        System.out.println(message);
    }
}
```

The resulting output from the invocation of the `execute()` method is as follows:

```
The userDao argument is required.
```

With regard to internationalization ("i18n"), Spring's various `MessageSource` implementations follow the same locale resolution and fallback rules as the standard JDK `ResourceBundle`. In short, and continuing with the example `messageSource` defined previously, if you want to resolve messages against the British (`en-GB`) locale, you would create files called `format_en_GB.properties`, `exceptions_en_GB.properties`, and `windows_en_GB.properties`, respectively.

Typically, locale resolution is managed by the surrounding environment of the application. In the following example, the locale against which (British) messages are resolved is specified manually:

```
# in exceptions_en_GB.properties
argument.required=Ebagum lad, the {0} argument is required, I say, required.
```

```
public static void main(final String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("argument.required",
        new Object [] {"userDao"}, "Required", Locale.UK);
    System.out.println(message);
}
```

The resulting output from the running of the above program is as follows:

```
Ebagum lad, the 'userDao' argument is required, I say, required.
```

You can also use the `MessageSourceAware` interface to acquire a reference to any `MessageSource` that has been defined. Any bean that is defined in an `ApplicationContext` that implements the `MessageSourceAware` interface is injected with the application context's `MessageSource` when the bean is created and configured.

NOTE As an alternative to `ResourceBundleMessageSource`, Spring provides a `ReloadableResourceBundleMessageSource` class. This variant supports the same bundle file format but is more flexible than the standard JDK based `ResourceBundleMessageSource` implementation. In particular, it allows for reading files from any Spring resource location (not only from the classpath) and supports hot reloading of bundle property files (while efficiently caching them in between). See the {api-spring-framework}/context/support/ReloadableResourceBundleMessageSource.html[`ReloadableResourceBundleMessageSource`] javadoc for details.

Standard and Custom Events

Event handling in the `ApplicationContext` is provided through the `ApplicationEvent` class and the `ApplicationListener` interface. If a bean that implements the `ApplicationListener` interface is deployed into the context, every time an `ApplicationEvent` gets published to the `ApplicationContext`, that bean is notified. Essentially, this is the standard Observer design pattern.

TIP As of Spring 4.2, the event infrastructure has been significantly improved and offers an annotation-based model as well as the ability to publish any arbitrary event (that is, an object that does not necessarily extend from `ApplicationEvent`). When such an object is published, we wrap it in an event for you.

The following table describes the standard events that Spring provides:

Table 7. Built-in Events

Event	Explanation
ContextRefreshedEvent	Published when the ApplicationContext is initialized or refreshed (for example, by using the refresh() method on the ConfigurableApplicationContext interface). Here, "initialized" means that all beans are loaded, post-processor beans are detected and activated, singletons are pre-instantiated, and the ApplicationContext object is ready for use. As long as the context has not been closed, a refresh can be triggered multiple times, provided that the chosen ApplicationContext actually supports such "hot" refreshes. For example, XmlWebApplicationContext supports hot refreshes, but GenericApplicationContext does not.
ContextStartedEvent	Published when the ApplicationContext is started by using the start() method on the ConfigurableApplicationContext interface. Here, "started" means that all Lifecycle beans receive an explicit start signal. Typically, this signal is used to restart beans after an explicit stop, but it may also be used to start components that have not been configured for autostart (for example, components that have not already started on initialization).
ContextStoppedEvent	Published when the ApplicationContext is stopped by using the stop() method on the ConfigurableApplicationContext interface. Here, "stopped" means that all Lifecycle beans receive an explicit stop signal. A stopped context may be restarted through a start() call.
ContextClosedEvent	Published when the ApplicationContext is closed by using the close() method on the ConfigurableApplicationContext interface. Here, "closed" means that all singleton beans are destroyed. A closed context reaches its end of life. It cannot be refreshed or restarted.
RequestHandledEvent	A web-specific event telling all beans that an HTTP request has been serviced. This event is published after the request is complete. This event is only applicable to web applications that use Spring's DispatcherServlet.

You can also create and publish your own custom events. The following example shows a simple class that extends Spring's ApplicationEvent base class:

```
public class BlackListEvent extends ApplicationEvent {

    private final String address;
    private final String content;

    public BlackListEvent(Object source, String address, String content) {
        super(source);
        this.address = address;
        this.content = content;
    }

    // accessor and other methods...
}
```

To publish a custom `ApplicationEvent`, call the `publishEvent()` method on an `ApplicationEventPublisher`. Typically, this is done by creating a class that implements `ApplicationEventPublisherAware` and registering it as a Spring bean. The following example shows such a class:

```
public class EmailService implements ApplicationEventPublisherAware {

    private List<String> blackList;
    private ApplicationEventPublisher publisher;

    public void setBlackList(List<String> blackList) {
        this.blackList = blackList;
    }

    public void setApplicationEventPublisher(ApplicationEventPublisher publisher)
{
        this.publisher = publisher;
    }

    public void sendEmail(String address, String content) {
        if (blackList.contains(address)) {
            publisher.publishEvent(new BlackListEvent(this, address, content));
            return;
        }
        // send email...
    }
}
```

At configuration time, the Spring container detects that `EmailService` implements `ApplicationEventPublisherAware` and automatically calls `setApplicationEventPublisher()`. In reality, the parameter passed in is the Spring container itself. You are interacting with the application context through its ApplicationEventPublisher interface.

To receive the custom ApplicationEvent, you can create a class that implements ApplicationListener and register it as a Spring bean. The following example shows such a class:

```
public class BlackListNotifier implements ApplicationListener<BlackListEvent> {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    public void onApplicationEvent(BlackListEvent event) {
        // notify appropriate parties via notificationAddress...
    }
}
```

Notice that ApplicationListener is generically parameterized with the type of your custom event (BlackListEvent in the preceding example). This means that the onApplicationEvent() method can remain type-safe, avoiding any need for downcasting. You can register as many event listeners as you wish, but note that, by default, event listeners receive events synchronously. This means that the publishEvent() method blocks until all listeners have finished processing the event. One advantage of this synchronous and single-threaded approach is that, when a listener receives an event, it operates inside the transaction context of the publisher if a transaction context is available. If another strategy for event publication becomes necessary, See the javadoc for Spring's {api-spring-framework}/context/event/ApplicationEventMulticaster.html[ApplicationEventMulticaster] interface.

The following example shows the bean definitions used to register and configure each of the classes above:

```
<bean id="emailService" class="example.EmailService">
    <property name="blackList">
        <list>
            <value>known.spammer@example.org</value>
            <value>known.hacker@example.org</value>
            <value>john.doe@example.org</value>
        </list>
    </property>
</bean>

<bean id="blackListNotifier" class="example.BlackListNotifier">
    <property name="notificationAddress" value="blacklist@example.org"/>
</bean>
```

Putting it all together, when the `sendEmail()` method of the `emailService` bean is called, if there are any email messages that should be blacklisted, a custom event of type `BlackListEvent` is published. The `blackListNotifier` bean is registered as an `ApplicationListener` and receives the `BlackListEvent`, at which point it can notify appropriate parties.

NOTE	Spring's eventing mechanism is designed for simple communication between Spring beans within the same application context. However, for more sophisticated enterprise integration needs, the separately maintained Spring Integration project provides complete support for building lightweight, pattern-oriented, event-driven architectures that build upon the well-known Spring programming model.

Annotation-based Event Listeners

As of Spring 4.2, you can register an event listener on any public method of a managed bean by using the `EventListener` annotation. The `BlackListNotifier` can be rewritten as follows:

```java
public class BlackListNotifier {

    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    @EventListener
    public void processBlackListEvent(BlackListEvent event) {
        // notify appropriate parties via notificationAddress...
    }
}
```

The method signature once again declares the event type to which it listens, but, this time, with a flexible name and without implementing a specific listener interface. The event type can also be narrowed through generics as long as the actual event type resolves your generic parameter in its implementation hierarchy.

If your method should listen to several events or if you want to define it with no parameter at all, the event types can also be specified on the annotation itself. The following example shows how to do so:

```java
@EventListener({ContextStartedEvent.class, ContextRefreshedEvent.class})
public void handleContextStart() {
    ...
}
```

It is also possible to add additional runtime filtering by using the `condition` attribute of the annotation that defines a `SpEL` expression , which should match to actually invoke the method for a particular event.

The following example shows how our notifier can be rewritten to be invoked only if the `content` attribute of the event is equal to `my-event`:

```
@EventListener(condition = "#blEvent.content == 'my-event'")
public void processBlackListEvent(BlackListEvent blEvent) {
    // notify appropriate parties via notificationAddress...
}
```

Each `SpEL` expression evaluates against a dedicated context. The following table lists the items made available to the context so that you can use them for conditional event processing:

Table 8. Event SpEL available metadata

Name	Location	Description	Example
Event	root object	The actual `ApplicationEvent`.	`#root.event`
Arguments array	root object	The arguments (as array) used for invoking the target.	`#root.args[0]`
Argument name	evaluation context	The name of any of the method arguments. If, for some reason, the names are not available (for example, because there is no debug information), the argument names are also available under the `#a<#arg>` where `#arg` stands for the argument index (starting from 0).	`#blEvent` or `#a0` (you can also use `#p0` or `#p<#arg>` notation as an alias)

Note that `#root.event` gives you access to the underlying event, even if your method signature actually refers to an arbitrary object that was published.

If you need to publish an event as the result of processing another event, you can change the method signature to return the event that should be published, as the following example shows:

```
@EventListener
public ListUpdateEvent handleBlackListEvent(BlackListEvent event) {
    // notify appropriate parties via notificationAddress and
    // then publish a ListUpdateEvent...
}
```

NOTE This feature is not supported for asynchronous listeners.

This new method publishes a new `ListUpdateEvent` for every `BlackListEvent` handled by the method above. If you need to publish several events, you can return a `Collection` of events instead.

Asynchronous Listeners

If you want a particular listener to process events asynchronously, you can reuse the regular `@Async` support. The following example shows how to do so:

```
@EventListener
@Async
public void processBlackListEvent(BlackListEvent event) {
    // BlackListEvent is processed in a separate thread
}
```

Be aware of the following limitations when using asynchronous events:

- If the event listener throws an `Exception`, it is not propagated to the caller See `AsyncUncaughtExceptionHandler` for more details.
- Such event listener cannot send replies. If you need to send another event as the result of the processing, inject {api-spring-framework}/aop/interceptor/AsyncUncaughtExceptionHandler.html[`ApplicationEventPublisher`] to send the event manually.

Ordering Listeners

If you need one listener to be invoked before another one, you can add the `@Order` annotation to the method declaration, as the following example shows:

```
@EventListener
@Order(42)
public void processBlackListEvent(BlackListEvent event) {
    // notify appropriate parties via notificationAddress...
}
```

Generic Events

You can also use generics to further define the structure of your event. Consider using an `EntityCreatedEvent<T>` where `T` is the type of the actual entity that got created. For example, you can create the following listener definition to receive only `EntityCreatedEvent` for a `Person`:

```
@EventListener
public void onPersonCreated(EntityCreatedEvent<Person> event) {
    ...
}
```

Due to type erasure, this works only if the event that is fired resolves the generic parameters on which the event listener filters (that is, something like `class PersonCreatedEvent extends EntityCreatedEvent<Person> { … }`).

In certain circumstances, this may become quite tedious if all events follow the same structure (as should be the case for the event in the preceding example). In such a case, you can implement `ResolvableTypeProvider` to guide the framework beyond what the runtime environment provides. The following event shows how to do so:

```
public class EntityCreatedEvent<T> extends ApplicationEvent implements
ResolvableTypeProvider {

    public EntityCreatedEvent(T entity) {
        super(entity);
    }

    @Override
    public ResolvableType getResolvableType() {
        return ResolvableType.forClassWithGenerics(getClass(), ResolvableType
.forInstance(getSource()));
    }
}
```

TIP This works not only for `ApplicationEvent` but any arbitrary object that you send as an event.

Convenient Access to Low-level Resources

For optimal usage and understanding of application contexts, you should familiarize yourself with Spring’s `Resource` abstraction, as described in [resources].

An application context is a `ResourceLoader`, which can be used to load `Resource` objects. A `Resource` is essentially a more feature rich version of the JDK `java.net.URL` class. In fact, the implementations of the `Resource` wrap an instance of `java.net.URL`, where appropriate. A `Resource` can obtain low-level resources from almost any location in a transparent fashion, including from the classpath, a filesystem location, anywhere describable with a standard URL, and some other variations. If the resource location string is a simple path without any special prefixes, where those resources come from is specific and appropriate to the actual application context type.

You can configure a bean deployed into the application context to implement the special callback interface, `ResourceLoaderAware`, to be automatically called back at initialization time with the application context itself passed in as the `ResourceLoader`. You can also expose properties of type `Resource`, to be used to access static resources. They are injected into it like any other properties. You can specify those `Resource` properties as simple `String` paths and rely on automatic conversion from those text strings to actual `Resource` objects when the bean is deployed.

The location path or paths supplied to an `ApplicationContext` constructor are actually resource strings and, in simple form, are treated appropriately according to the specific context implementation. For example `ClassPathXmlApplicationContext` treats a simple location path as a classpath location. You can also use location paths (resource strings) with special prefixes to force loading of definitions from the classpath or a URL, regardless of the actual context type.

Convenient ApplicationContext Instantiation for Web Applications

You can create `ApplicationContext` instances declaratively by using, for example, a `ContextLoader`. Of course, you can also create `ApplicationContext` instances programmatically by using one of the `ApplicationContext` implementations.

You can register an `ApplicationContext` by using the `ContextLoaderListener`, as the following example shows:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-
value>
</context-param>

<listener>
    <listener-class>
org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

The listener inspects the `contextConfigLocation` parameter. If the parameter does not exist, the listener uses `/WEB-INF/applicationContext.xml` as a default. When the parameter does exist, the listener separates the `String` by using predefined delimiters (comma, semicolon, and whitespace) and uses the values as locations where application contexts are searched. Ant-style path patterns are supported as well. Examples are `/WEB-INF/*Context.xml` (for all files with names that end with `Context.xml` and that reside in the `WEB-INF` directory) and `/WEB-INF/**/*Context.xml` (for all such files in any subdirectory of `WEB-INF`).

Deploying a Spring `ApplicationContext` as a Java EE RAR File

It is possible to deploy a Spring `ApplicationContext` as a RAR file, encapsulating the context and all of its required bean classes and library JARs in a Java EE RAR deployment unit. This is the equivalent of bootstrapping a stand-alone `ApplicationContext` (only hosted in Java EE environment) being able to access the Java EE servers facilities. RAR deployment is a more natural alternative to a scenario of deploying a headless WAR file — in effect, a WAR file without any HTTP entry points that is used only for bootstrapping a Spring `ApplicationContext` in a Java EE environment.

RAR deployment is ideal for application contexts that do not need HTTP entry points but rather consist only of message endpoints and scheduled jobs. Beans in such a context can use application server resources such as the JTA transaction manager and JNDI-bound JDBC `DataSource` instances and JMS `ConnectionFactory` instances and can also register with the platform's JMX server — all through Spring's standard transaction management and JNDI and JMX support facilities. Application components can also interact with the application server's JCA `WorkManager` through Spring's `TaskExecutor` abstraction.

See the javadoc of the {api-spring-framework}/jca/context/SpringContextResourceAdapter.html[`SpringContextResourceAdapter`] class for the configuration details involved in RAR deployment.

For a simple deployment of a Spring ApplicationContext as a Java EE RAR file:

1. Package all application classes into a RAR file (which is a standard JAR file with a different file extension). .Add all required library JARs into the root of the RAR archive. .Add a `META-INF/ra.xml` deployment descriptor (as shown in the {api-spring-framework}/jca/context/SpringContextResourceAdapter.html[javadoc for `SpringContextResourceAdapter`]) and the corresponding Spring XML bean definition file(s) (typically `META-INF/applicationContext.xml).
2. Drop the resulting RAR file into your application server's deployment directory.

NOTE

Such RAR deployment units are usually self-contained. They do not expose components to the outside world, not even to other modules of the same application. Interaction with a RAR-based `ApplicationContext` usually occurs through JMS destinations that it shares with other modules. A RAR-based `ApplicationContext` may also, for example, schedule some jobs or react to new files in the file system (or the like). If it needs to allow synchronous access from the outside, it could (for example) export RMI endpoints, which may be used by other application modules on the same machine.

The `BeanFactory`

The `BeanFactory` API provides the underlying basis for Spring's IoC functionality. Its specific contracts are mostly used in integration with other parts of Spring and related third-party frameworks, and its DefaultListableBeanFactory implementation is a key delegate within the higher-level GenericApplicationContext container.

BeanFactory and related interfaces (such as BeanFactoryAware, InitializingBean, DisposableBean) are important integration points for other framework components. By not requiring any annotations or even reflection, they allow for very efficient interaction between the container and its components. Application-level beans may use the same callback interfaces but typically prefer declarative dependency injection instead, either through annotations or through programmatic configuration.

Note that the core BeanFactory API level and its DefaultListableBeanFactory implementation do not make assumptions about the configuration format or any component annotations to be used. All of these flavors come in through extensions (such as XmlBeanDefinitionReader and AutowiredAnnotationBeanPostProcessor) and operate on shared BeanDefinition objects as a core metadata representation. This is the essence of what makes Spring's container so flexible and extensible.

BeanFactory or ApplicationContext?

This section explains the differences between the BeanFactory and ApplicationContext container levels and the implications on bootstrapping.

You should use an ApplicationContext unless you have a good reason for not doing so, with GenericApplicationContext and its subclass AnnotationConfigApplicationContext as the common implementations for custom bootstrapping. These are the primary entry points to Spring's core container for all common purposes: loading of configuration files, triggering a classpath scan, programmatically registering bean definitions and annotated classes, and (as of 5.0) registering functional bean definitions.

Because an ApplicationContext includes all the functionality of a BeanFactory, it is generally recommended over a plain BeanFactory, except for scenarios where full control over bean processing is needed. Within an ApplicationContext (such as the GenericApplicationContext implementation), several kinds of beans are detected by convention (that is, by bean name or by bean type — in particular, post-processors), while a plain DefaultListableBeanFactory is agnostic about any special beans.

For many extended container features, such as annotation processing and AOP proxying, the BeanPostProcessor extension point is essential. If you use only a plain DefaultListableBeanFactory, such post-processors do not get detected and activated by default. This situation could be confusing, because nothing is actually wrong with your bean configuration. Rather, in such a scenario, the container needs to be fully bootstrapped through additional setup.

The following table lists features provided by the BeanFactory and ApplicationContext interfaces and implementations.

Table 9. Feature Matrix

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	Yes	Yes

Feature	`BeanFactory`	`ApplicationContext`
Integrated lifecycle management	No	Yes
Automatic `BeanPostProcessor` registration	No	Yes
Automatic `BeanFactoryPostProcessor` registration	No	Yes
Convenient `MessageSource` access (for internalization)	No	Yes
Built-in `ApplicationEvent` publication mechanism	No	Yes

To explicitly register a bean post-processor with a `DefaultListableBeanFactory`, you need to programmatically call `addBeanPostProcessor`, as the following example shows:

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
// populate the factory with bean definitions

// now register any needed BeanPostProcessor instances
factory.addBeanPostProcessor(new AutowiredAnnotationBeanPostProcessor());
factory.addBeanPostProcessor(new MyBeanPostProcessor());

// now start using the factory
```

To apply a `BeanFactoryPostProcessor` to a plain `DefaultListableBeanFactory`, you need to call its `postProcessBeanFactory` method, as the following example shows:

```
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(new FileSystemResource("beans.xml"));

// bring in some property values from a Properties file
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));

// now actually do the replacement
cfg.postProcessBeanFactory(factory);
```

In both cases, the explicit registration steps are inconvenient, which is why the various `ApplicationContext` variants are preferred over a plain `DefaultListableBeanFactory` in Spring-backed applications, especially when relying on `BeanFactoryPostProcessor` and `BeanPostProcessor` instances for extended container functionality in a typical enterprise setup.

NOTE

An `AnnotationConfigApplicationContext` has all common annotation post-processors registered and may bring in additional processors underneath the covers through configuration annotations, such as `@EnableTransactionManagement`. At the abstraction level of Spring's annotation-based configuration model, the notion of bean post-processors becomes a mere internal container detail.