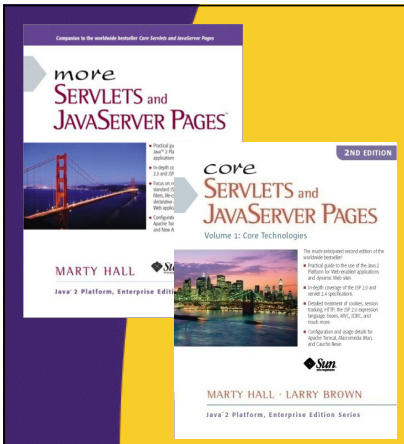
The Spring Framework: Core Capabilities Part 1

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/spring.html

Customized Java EE Training: http://courses.coreservlets.com/
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.



For live Spring & Hibernate training, see courses at http://courses.coreservlets.com/.

Taught by the experts that brought you this tutorial. Available at public venues, or customized versions can be held on-site at <u>your</u> organization.

- Courses developed and taught by Marty Hall
  - Java 5, Java 6, intermediate/beginning servlets/JSP, advanced servlets/JSP, Struts, JSF, Ajax, GWT, custom mix of topics
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Spring, Hibernate/JPA, EJB3, Ruby/Rails

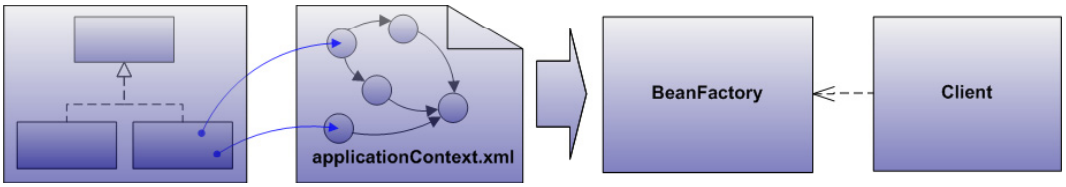
Contact hall@coreservlets.com for details

Topics in This Section

- Spring IoC container
- Interface-oriented development
- Spring framework composition
- Spring container instantiation
- Spring bean definitions

Spring IoC Container

Spring IoC Container



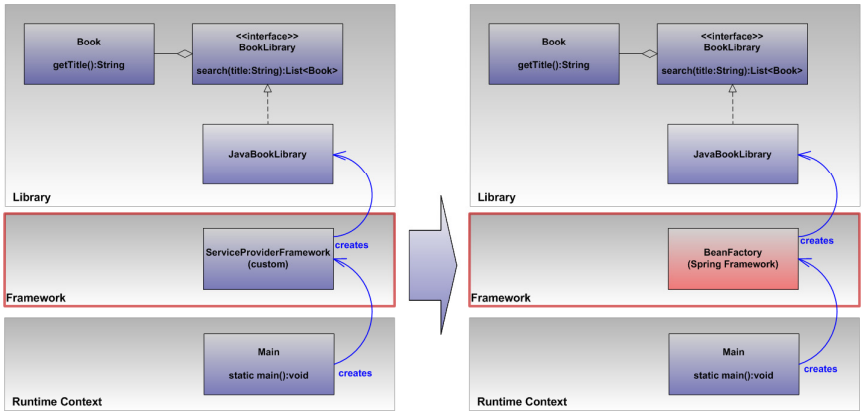
- **Fundamental features**
  - Registration system (**`applicationContext.xml`**)
    - Complex initialization
    - Object creation
    - Dependency injection
    - Application configuration
  - Access API
    - **`BeanFactory#getBean`**

Spring IoC Container Relevance

- **Inversion of control**
  - Moves object creation and dependency resolution responsibilities out of business logic
  - Enables business logic to be portable
  - Allows components to be reconfigured with minimal effort
- **Loose coupling**
  - Clients are insulated from implementations
  - Implementation technicalities are invisible to clients
  - Clients are only aware of the interface contract and unaware of:
    - Concrete type selections
    - Concrete type initialization mechanisms

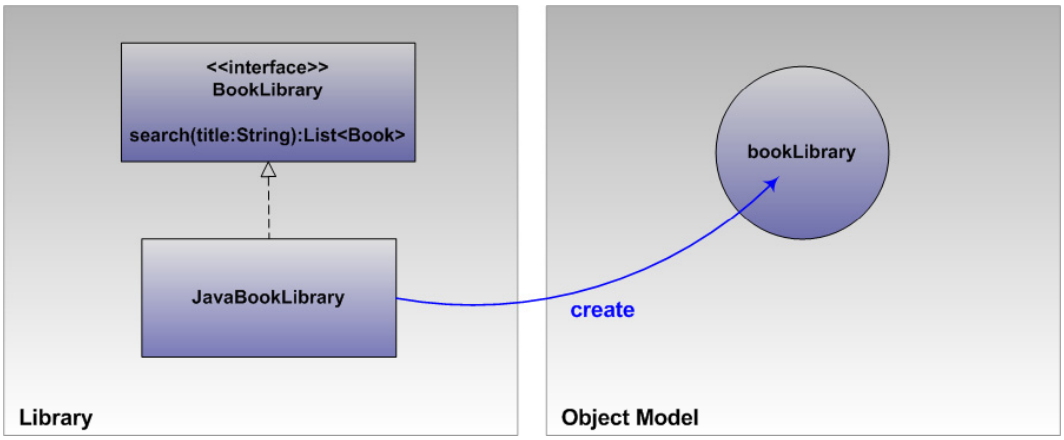
Spring IoC Container and POJO Instantiation

- **Integrate Spring IoC container**
  - Replace custom ServiceProviderFramework with BeanFactory
- **Delegate to Spring IoC container**
  - Object creation



Spring IoC Objective

- **Create a `BookLibrary` object based on the `JavaBookLibrary` class**



Spring IoC Process

- **Create an XML file conforming to `spring-beans.xsd`**
  - Name the file
    - Conventional name is `applicationContext.xml`
    - However, any name will suffice
  - Place the file in an accessible location
    - For example, in a filesystem directory which will be accessible from the classpath
  - Register objects
    - Objects are registered by declaring `bean` XML elements
    - Conventional approach is to use `bean` attributes: `id` and `class`
- **Access object(s) managed by the Spring IoC container**
  - Instantiate a `BeanFactory` implementation
  - Use interfaces such as `BeanFactory#getBean(...):Object`

Spring IoC Configuration

- **Object registration process**

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

  <bean id="bookLibrary" class="coreservlets.JavaBookLibrary"/>

</beans>
```

Registered name

Concrete type

Spring IoC BeanFactory

```
import org.springframework.beans.factory.*;
import org.springframework.context.support.*;

public class Main {
  public static void main(String[] args) {

    BeanFactory beanFactory =
      new ClassPathXmlApplicationContext(
        "/applicationContext.xml");

    BookLibrary service =
      (BookLibrary)beanFactory.getBean("bookLibrary");

    System.out.printf("Retrieved BookLibrary type: \"%s\"%n",
                      service.getClass().getSimpleName());
  }
}
```

Spring BeanFactory configuration

Spring BeanFactory access API

Interface type

Standard output

```
Retrieved BookLibrary type: "JavaBookLibrary"
```


Spring IoC Container Summary

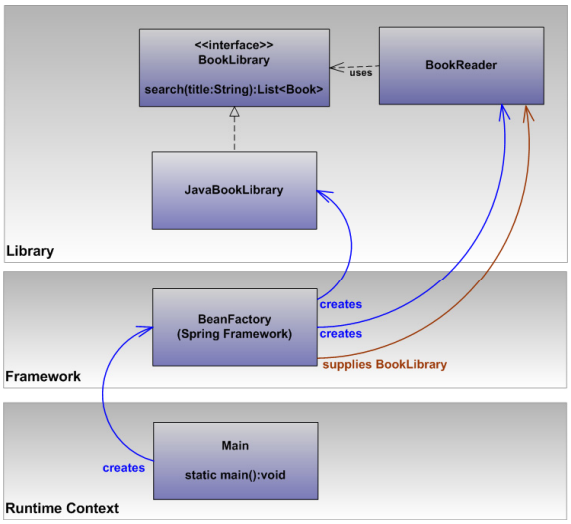
- **Loaded**
  - Bean definitions from **`applicationContext.xml`**
- **Created**
  - **`coreservlets.JavaBookLibrary`** instance
- **Registered**
  - Object under the name **`bookLibrary`**
- **Requested**
  - A **`BookLibrary`** object from the Spring IoC container using the access API, **`BeanFactory#getBean(…)`**
- **Received**
  - An instance of **`JavaBookLibrary`**, a **`BookLibrary`** implementation instance

Model Analysis

- **Dynamic implementation choices**
  - Implementation types are excluded from the program
- **Portable model configuration**
  - Object model configuration is encapsulated within the framework
- **Flexible model configuration**
  - Object model configuration is a declarative system based on **`spring-beans.xsd`**

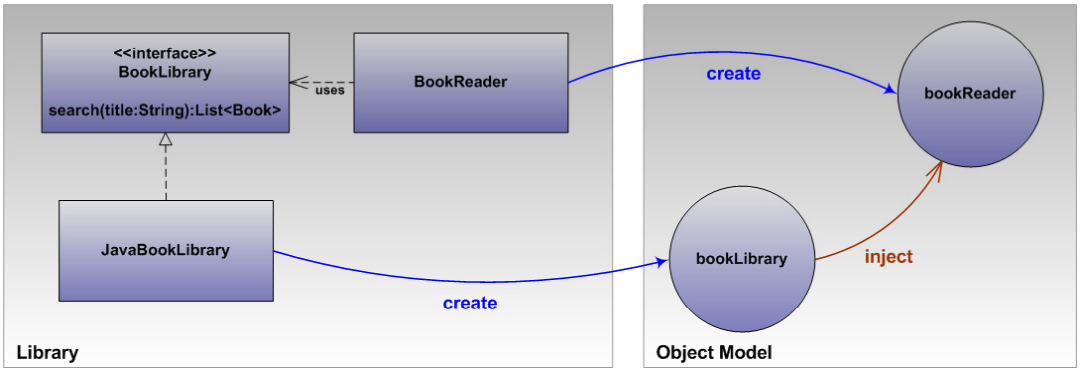
Spring IoC Container and Dependency Injection

- **Delegate to Spring IoC container**
  - Object instantiation
  - **Dependency injection**



Spring IoC Objective

- Create a `BookLibrary` object based on the `JavaBookLibrary` class
- Create a `BookReader` object based on the `BookReader` class
- Inject the `BookLibrary` object into `BookReader`



Spring Process

- Create an XML file conforming to `spring-beans.xsd`
  - Name the file
  - Place the file in an accessible location such as the classpath
  - Register objects using XML `bean` elements
  - Add bean dependency injection instructions
    - Add references to beans, values, collections, or configuration properties
    - Also known as "wiring" the application
- Access object(s) managed by the Spring IoC container
  - Instantiate a `BeanFactory` implementation
  - Use interfaces such as `BeanFactory#getBean(...):Object`

Spring XML Configuration

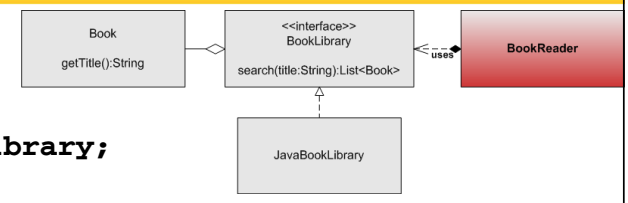
- **Object registration process**

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

  <bean id="bookLibrary" class="coreservlets.JavaBookLibrary"/>

  <bean id="bookReader" class="coreservlets.BookReader">
    <constructor-arg ref="bookLibrary"/>
  </bean>

</beans>
```

Referenceable bean

Bean reference

BookReader Constructor




```
public class BookReader {

  private BookLibrary bookLibrary;

  public BookReader(BookLibrary bookLibrary) {
    this.bookLibrary = bookLibrary;
  }

  public List<Book> read() {
    List<Book> books = bookLibrary.search("Java");
    for(Book book : books){
      System.out.printf("Reading: %s%n", book);
    }
    return books;
  }
}
```

Dependency injection interface

Spring BeanFactory

```
public class Main {
  public static void main(String[] args) {

    BeanFactory beanFactory =
      new ClassPathXmlApplicationContext(
        "/applicationContext.xml");

    BookReader client = (BookReader)
      beanFactory.getBean("bookReader");

    List<Book> books = client.read();

    System.out.printf("Client read: %s books%n",
                      books.size());  }
}
```

Spring BeanFactory configuration

Spring BeanFactory access API

Standard output

```
Reading: Core Servlets and JavaServer Pages
Reading: More Servlets and JavaServer Pages
Client read: 2 books
```


Spring IoC Container Summary

- **Loaded**
  - Bean definitions from **`applicationContext.xml`**
- **Created**
  - **`coreservlet.JavaBookLibrary`** instance
  - **`coreservlet.BookReader`** instance
- **Registered**
  - Object under the name **`bookLibrary`**
  - Object under the name **`bookReader`**
- **Injected dependency**
  - **`BookLibrary`** implementation instance, **`JavaBookLibrary`**, into the **`BookReader`** object

Spring IoC Container Summary Continued

- **Requested**
  - A **`BookReader`** object from the Spring IoC container using the access API, **`BeanFactory#getBean(…)`**
- **Received**
  - An instance of **`BookReader`**
- **Used**
  - The **`BookLibrary`** dependency was previously fulfilled during the injection step
  - **`BookReader`** used **`BookLibrary`** interfaces to access and read **`Book`** information

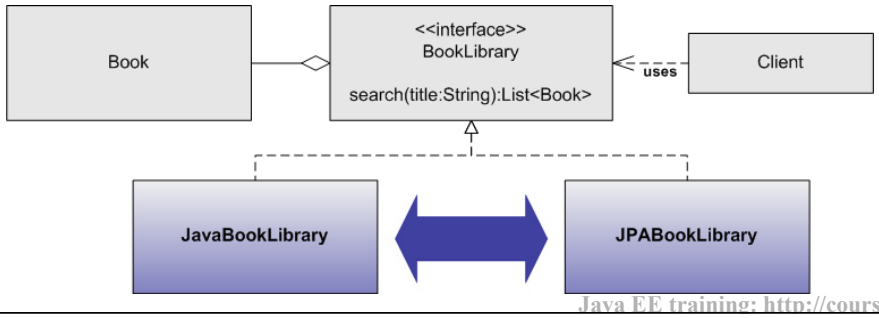
Model Analysis

- **Dynamic implementation choices**
  - Implementation types are excluded from the program
  - Decoupled design allows new types to be configured into the program without having to recompile
- **Portable model configuration**
  - Object model configuration is encapsulated within the framework
- **Flexible model configuration**
  - Object model configuration is a declarative system based on **`spring-beans.xsd`**
  - **`BeanFactory`** clients are also decoupled from the bean types accessed from the container

Interface-Oriented Development

Interface-Oriented Development

- **Take advantage of type-polymorphism**
  - Flexible architecture
  - Tolerant to changes
  - Enables new capabilities with minimal effort
- **Commit to interfaces, not implementations**
  - Included, but not limited to, compiled interactions
  - Declarative interfaces



Interface-Oriented Example

```
<bean id="bookLibrary"
    class="coreservlets.JavaBookLibrary"/>

<bean id="bookReader"
      class="coreservlets.BookReader" >
  <constructor-arg ref="bookLibrary"/>
</bean>


public class BookReader {

  private BookLibrary bookLibrary;

  public BookReader(BookLibrary bookLibrary) {
    this.bookLibrary = bookLibrary;
  }
  ...
}
```

Interface type

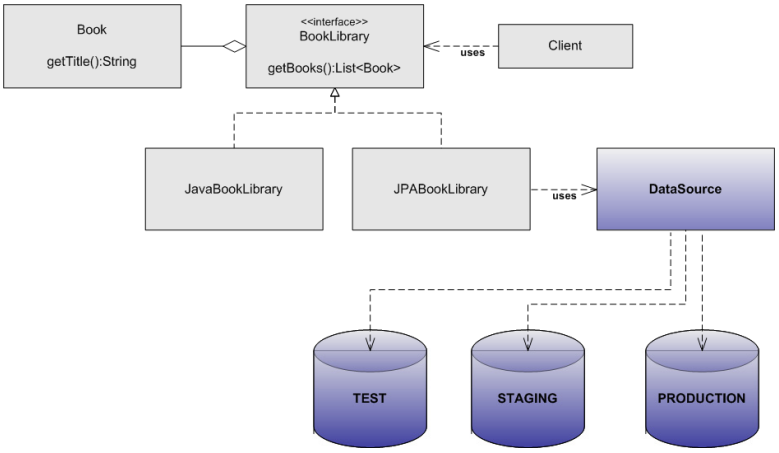
Interface-Oriented Example

```
<bean id="bookLibrary"
      class="coreservlets.JavaBookLibrary" />



public class Main {
  public static void main(String[] args) {
    ...
    BookLibrary service =
      (BookLibrary)beanFactory.getBean("bookLibrary");

  }
}
```

Interface type

Service Abstraction

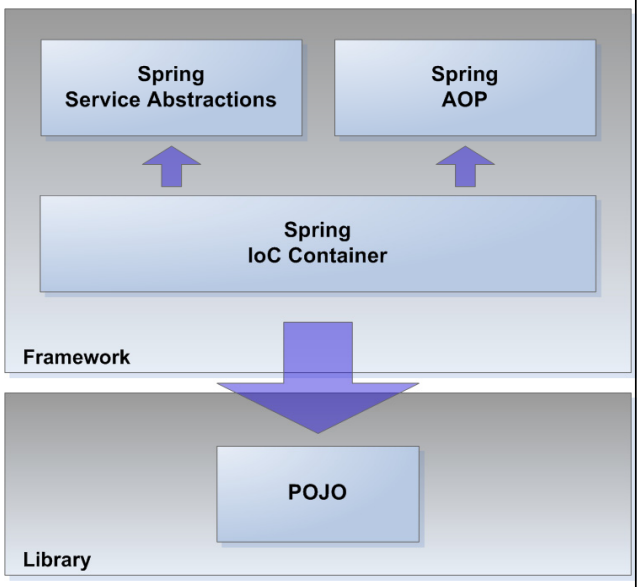
- Abstract elements such as third-party APIs and infrastructure
- Decouple business logic enabling portability



Spring Framework

Spring Framework Composition

- **IoC Container**
  - Core libraries
    - **`spring-core`**
    - **`spring-beans`**
  - Integration extensions
    - **`spring-context`**
    - **`spring-web`**



Spring Framework Modules

- **Aspect-Oriented Programming**
  - **`spring-aop`**
- **Service abstractions**
  - JDBC templates and transaction management
    - **`spring-jdbc`**
    - **`spring-tx`**
  - O/R mapping frameworks
    - **`spring-orm`**
  - JMS
    - **`spring-jms`**
  - JEE, EJB, JMX, JNDI, etc…
    - **`spring-context`**
  - Java Mail, Quartz, etc…
    - **`spring-context-support`**

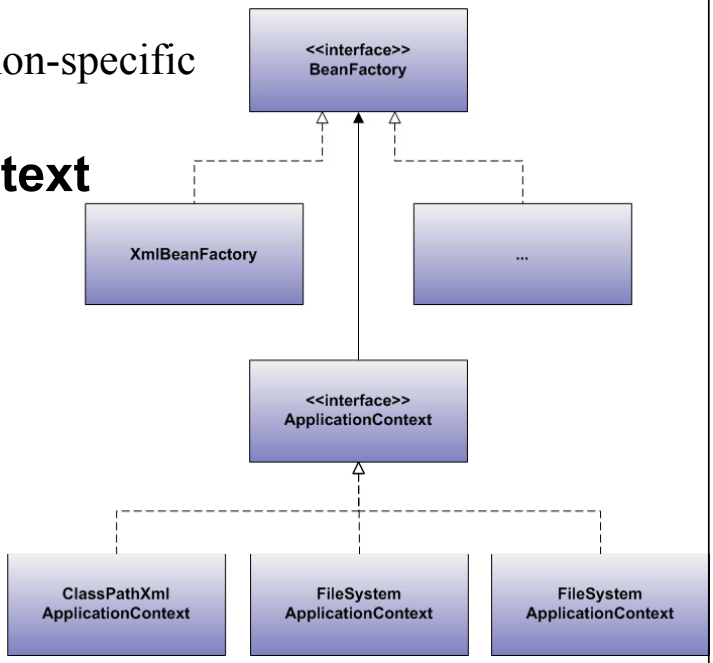
Spring Framework Composition

- **Test support**
  - TestNG and JUnit, IoC container, and transaction management integration
    - `spring-test`
- **Web Application Framework**
  - Spring Web MVC, FreeMarker, and Jasper Reports
    - `spring-webmvc`

Spring `BeanFactory`

BeanFactory

- **BeanFactory**
  - Defines core but non-specific functionality
- **ApplicationContext**
  - Context resources
  - Bundle resources
  - Event listeners
  - Post processors



Coarse-Grained Interfaces

- **Covers typical integration scenarios**
- **Automated but implicit functionality**
- **Committed to specific integration strategies**

```
BeanFactory beanFactory =
      new ClassPathXmlApplicationContext(
      "/applicationContext.xml");

BeanFactory beanFactory =
      new FileSystemXmlApplicationContext(
      "/etc/app/applicationContext.xml");
```

Fine-Grained Interfaces

- **Fine integration control**
  - Bean configuration abstraction
  - I/O abstraction

```
BeanFactory beanFactory =
  new GenericApplicationContext();

XmlBeanDefinitionReader xmlReader =
  new XmlBeanDefinitionReader(beanFactory);

xmlReader.loadBeanDefinitions(
  new FileInputStream("/etc/applicationContext.xml"));

xmlReader.loadBeanDefinitions(
  new ClassPathResource("/applicationContext.xml"));
```

Parses and loads bean definitions

Integration with generic I/O interfaces

Integration with spring I/O abstractions

Servlet Integration

```
<web-app>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      WEB-INF/applicationContext.xml
      classpath:/applicationContext.xml
    </param-value>
  </context-param>
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
</web-app>
```

Example of Spring resource abstraction

JavaServer Faces Integration

- **Multiple variable and EL resolver implementations**
  - JSF 1.1
    - DelegatingVariableResolver
    - SpringBeanVariableResolver
  - JSF 1.2
    - SpringBeanFacesELResolver

```
<faces-config>
  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
  </application>
</faces-config>
```


JUnit Test Integration

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class BookLibraryTest {

  @Autowired
  public BookLibrary bookLibrary;

  @Test
  public void verifySearch(){ ... }

}

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"/applicationContext.xml"})
public class BookLibraryTest {
  ...
}
```

JUnit integration

Loads bean definitions defaults to applicationContext.xml

Loads bean definitions using explicit paths

JUnit Test Integration

```
@ContextConfiguration
public class BookLibraryTest
    extends AbstractJUnit4SpringContextTests {

  @Autowired
  public BookLibrary bookLibrary;

  @Test
  public void verifySearch(){
    ...
  }
}
```


Container Event Listener

- **Lifecycle event model**
  - Event listener interface
    `org.springframework.context.ApplicationListener`
  - Event objects
    `org.springframework.context.ApplicationEvent`
    - `ContextRefreshEvent`
    - `ContextStartedEvent`
    - `ContextStoppedEvent`
    - `ContextClosedEvent`
    - `RequestHandledEvent`
- **Registration model**
  - Enabled by registering new listener instances as container-managed beans

Container Event Listener Example

```
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;

public class SimpleApplicationListener
    implements ApplicationListener {

  public void onApplicationEvent(ApplicationEvent event) {
     System.out.printf("Event type: %s%n",
                       event.getClass().getSimpleName());
  }
}

<beans>
  <bean id="applicationListener"
        class="coreservlets.SimpleApplicationListener" />
</beans>
```


Container Event Listener Example

```
import org.springframework.beans.factory.*;
import org.springframework.context.*;
import org.springframework.context.support.*;
public class Main {
  public static void main(String[] args) {
    BeanFactory beanFactory = new
      ClassPathXmlApplicationContext(
        "/applicationContext.xml");

    ConfigurableApplicationContext configurableContext =
      (ConfigurableApplicationContext) beanFactory;
    configurableContext.start();
    configurableContext.stop();
    configurableContext.close();
  }
}
```

Instantiate container

Expose interfaces for triggering lifecycle events

Trigger lifecycle events

Standard output

```
Event type: ContextRefreshedEvent
Event type: ContextStartedEvent
Event type: ContextStoppedEvent
Event type: ContextClosedEvent
```

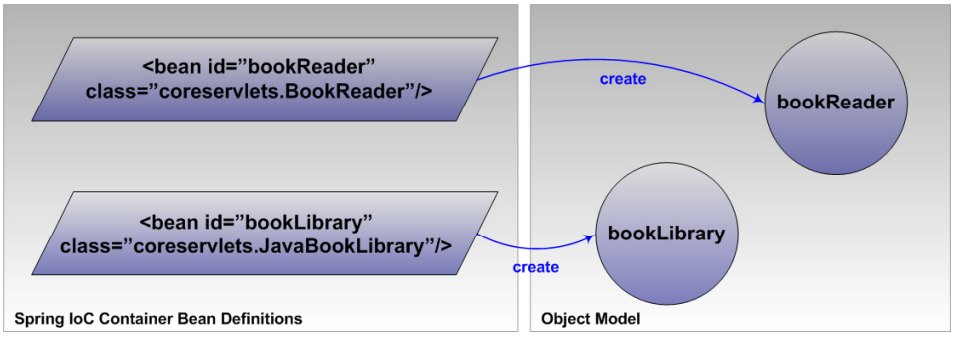
Bean Definition

Bean Definitions

- Default bean definition
- Factory bean
- Static factory method
- Abstract bean

Default Bean Definition

- **Create container-managed objects directly out of bean elements**
  - Standard bean definitions:
    - XML **`bean`** element. Child to document root, **`bean`**
  - Inner bean definitions
    - XML **`bean`** element. Child to **`property or constructor-arg`** elements



Default Bean Definition

- **Standard `bean` element**
  - Referenceable, aka collaborator, beans
- **Inner bean**
  - Anonymous
  - Prototype
  - Used to fulfill dependency injection settings
- **Invocation prerequisite(s)--one combination required**
  - **`class`**
  - **`class`** and **`factory-method`**
  - **`factory-bean`** and **`factory-method`**
  - **`abstract`** bean attribute set to **`true`**

Default Bean Definition

- **Optional `bean` properties**
  - **`id`** and **`name`**
    - defaults to a generated and unique value
  - **`scope`**
    - defaults to **`singleton`**
  - **`abstract/lazy-init/primary`**
    - defaults to false
  - **`autowire`**
    - defaults to no
  - **`dependency-check`** – defaults to none
  - **`parent/depends-on/autowire-candidate/init-method/destroy-method`**
    - defaults to **`null`**

Standard `bean`

- **Matches on a `Constructor`**
  - See java.lang.Class and java.lang.reflect
- **Invokes the `Constructor`**
  - Analogous to the **`new`** keyword, invoking a **`Constructor`** is the same as invoking the **`new`** operator

```
<bean id="bookLibrary"
      class="coreservlets.JavaBookLibrary">
  <!-- empty constructor arguments -->
</bean>

public class JavaBookLibrary {
  public JavaBookLibrary() {
    ...
  }
  ...
}
```

Constructor definition

Target constructor

Standard bean

- Matches on a `Constructor` by type

```
<bean id="bookLibrary"
      class="coreservlets.JavaBookLibrary">
</bean>

<bean id="bookReader" class="coreservlets.BookReader" >
  <constructor-arg ref="bookLibrary"/>
</bean>

public class BookReader {
  private BookLibrary bookLibrary;
  public BookReader(BookLibrary bookLibrary) {
    this.bookLibrary = bookLibrary;
  }
  ...
}
```

Bean
aka collaborator

Constructor definition
& bean reference

Constructor target

Inner bean Element

- Nested bean definitions

```
<bean id="bookReader" class="coreservlets.BookReader" >
  <constructor-arg>
    <bean class="coreservlets.JavaBookLibrary"/>
  </constructor-arg>
</bean>

public class BookReader {
  private BookLibrary bookLibrary;
  public BookReader(BookLibrary bookLibrary) {
    this.bookLibrary = bookLibrary;
  }
  ...
}
```

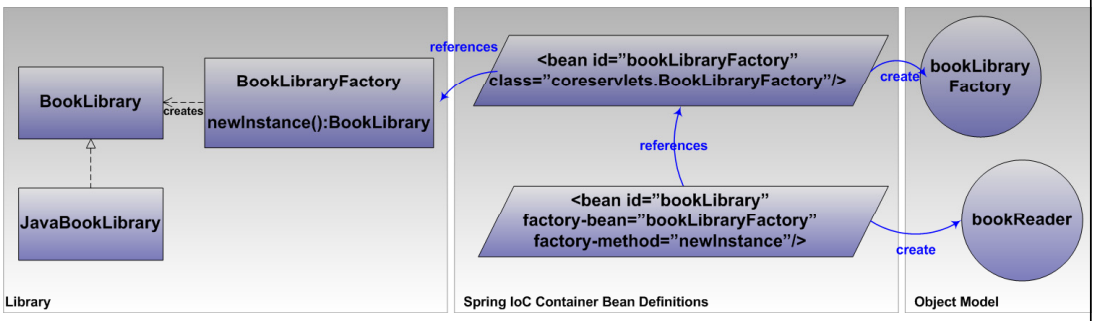
Anonymous inner bean

Constructor definition

Target constructor

Factory Bean

- **The invocation of a method on a container-managed factory bean to create other container-managed beans**
- **References to the target bean class are excluded**
  - The class information for the target bean is unnecessary



Factory Bean

- **Factory bean**
  - A container-managed bean responsible for instantiating other container-managed beans
  - Referenced by a **`bean`** element attribute, **`factory-bean`**
- **Factory method**
  - The factory bean exposes a method for instantiating the target bean type
  - Referenced by a **`bean`** element attribute, **`factory-method`**
- **Target bean**
  - The **`bean`** declaration uses **`factory-bean`** and **`factory-method`** to map its origin to the factory bean and factory method
  - The factory information substitutes the XML **`class`** attribute specification

Factory Bean Example

```
public class BookLibraryFactory {
   public BookLibrary newInstance() {
      return new JavaBookLibrary();
   }
}

<bean id="bookLibraryFactory"
      class="coreservlets.BookLibraryFactory"/>

<bean id="bookLibrary"
      factory-bean="bookLibraryFactory"
      factory-method="newInstance" />
```

Factory class

Factory method

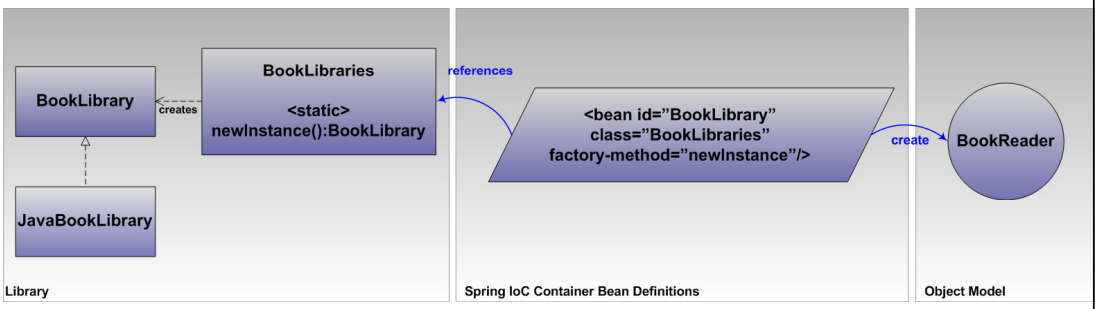
Factory bean

Factory bean reference

Factory method

Static Factory Method

- The invocation of a static factory method to create the container-managed bean
- Separate container-managed bean is not required
- The target bean references the class and method information as the origin



Static Factory Method Example

```
public class BookLibraries {
  public static BookLibrary newInstance() {
    return new JavaBookLibrary();
  }
}

<beans>
  <bean id="bookLibrary"
        class="coreservlets.BookLibraries"
        factory-method="newInstance"/>
</beans>
```

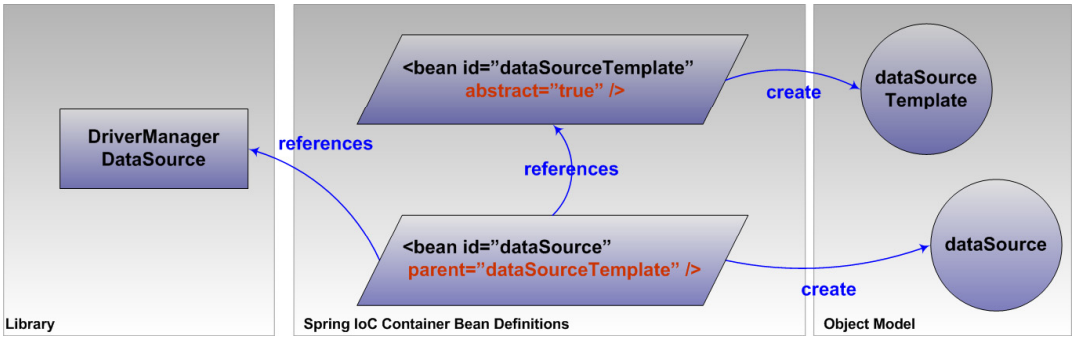
Static factory method

Class defining factory method

Static factory method

Abstract Bean

- **Bean templating facility**
- **Establishes default bean properties**
  - Constructor arguments, property setter parameters, bean lifecycle callbacks
- **Candidate classes for abstract bean definitions**



Abstract Bean Example

```
<bean id="dataSourceTemplate" abstract="true">
  <property name="url" value="jdbc:shinydb:" />
  <property name="driver" value="jdbc.ShinyDriver" />
  <property name="username" value="webapp" />
</bean>

<bean id="dataSource" class="DriverManagerDataSource">
      parent="dataSourceTemplate">
  <property name="password" value="orioles" />
</bean>
```

Abstract discriminator

Template bean reference

Wrapup

General Approach

- **Develop POJOs**
  - Implement creational patterns as needed
  - Assume the framework will accommodate the creational patterns required by the system
- **Plan on using a framework to manage the instantiation of all POJOs**
- **Create the XML bean definitions file**
  - This file is typically named **`applicationContext.xml`**
  - For large projects comprised of numerous modules, develop a predictable naming system for context files

General Approach (Continued)

- **Declare bean definitions with interdependencies**
  - Spring IoC provides comprehensive support for instantiation patterns
    - Constructor
    - Factory method
    - Static factory method
    - Template beans
- **Include the bean definitions file as part of the distribution**
  - This file will serve as a default, but optional, blueprint

Summary

- **Select a Spring IoC container integration approach to match the runtime context**
  - Broad integration options are available
    - Programmatic contexts
      - ClassPathXmlApplicationContext
      - FileSystemXmlApplicationContext
    - Platform-specific contexts
      - ContextLoaderListener
    - Test contexts
      - AbstractJUnit4SpringContextTests
  - ApplicationContext is favored over BeanFactory implementations
    - Improved defaults, integration extensions, and automated behavior

Questions?