



Introducing Spring

A Lightweight Java Container



Introducing Spring

- Spring was born from dislike of JEE
 - JEE can be viewed as cumbersome, awkward and intrusive
 - Especially for small to medium size projects
 - The creator of Spring (Ron Johnson) laid the foundations for a simpler approach in his book 'Expert One on One J2EE'
- Spring is a lightweight container
 - It provides a managed environment for your components
 - The environment aims to be as unobtrusive as possible
 - Your code is not tied to Spring interfaces, base classes etc...
 - Spring can be used inside, outside or instead of a JEE Server
 - BEA have enthusiastically adopted it as part of WebLogic



Spring History

- Expert One on One J2EE (November 2002)
- Spring 1.0 Milestone 1 (August 2003)
- Spring 1.0 (March 2004)
- Spring 1.1 (September 2004)
- Spring 1.2 (May 2005)
- Spring 2.0 Milestone 1 (December 2005)
- Spring 2.0 (October 2006)
 - Works with Java 1.3 but increasing numbers of features require Java 1.5 (e.g. parts of AOP support)
- Spring 2.5 (November 2007)

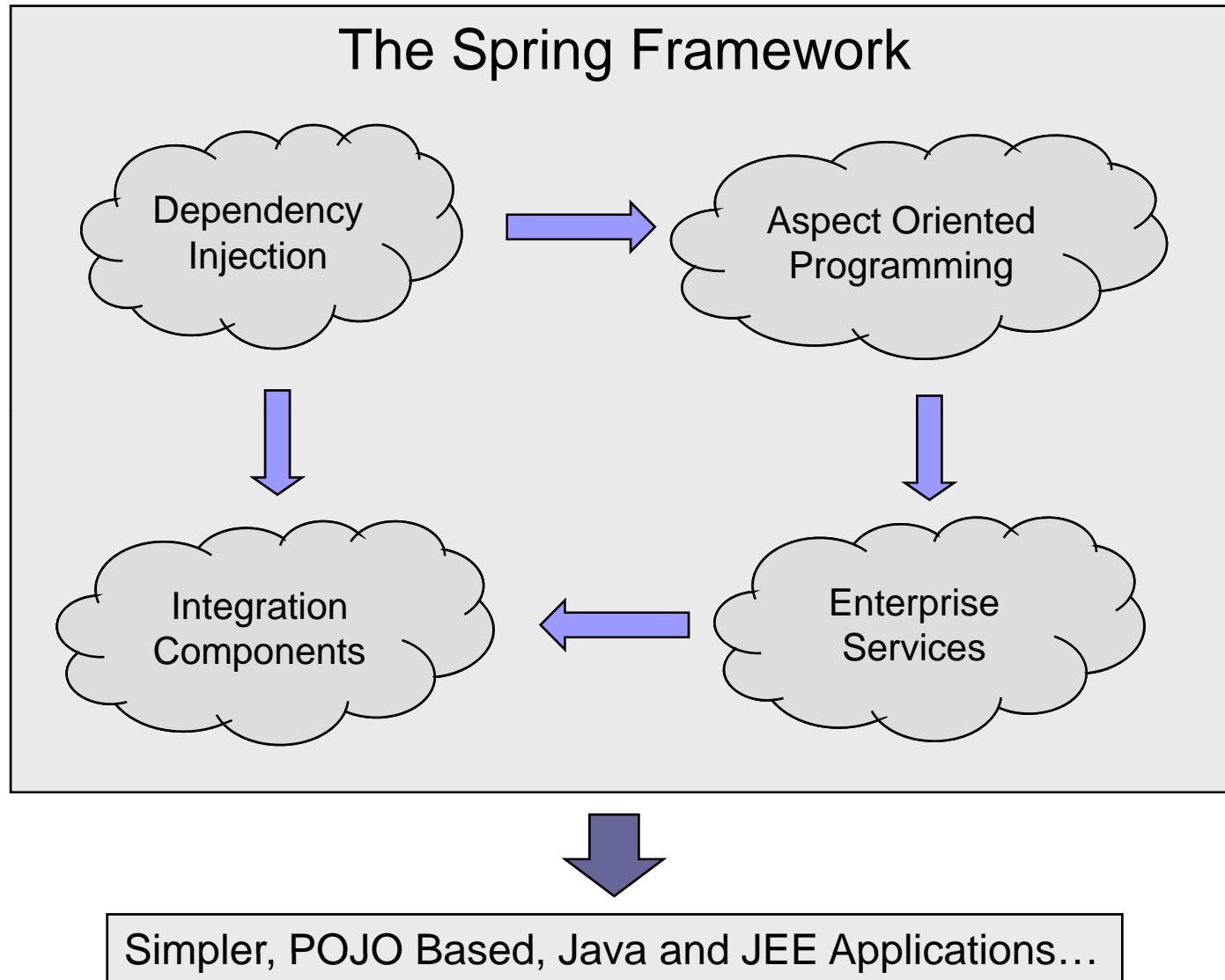


Key Features of Spring

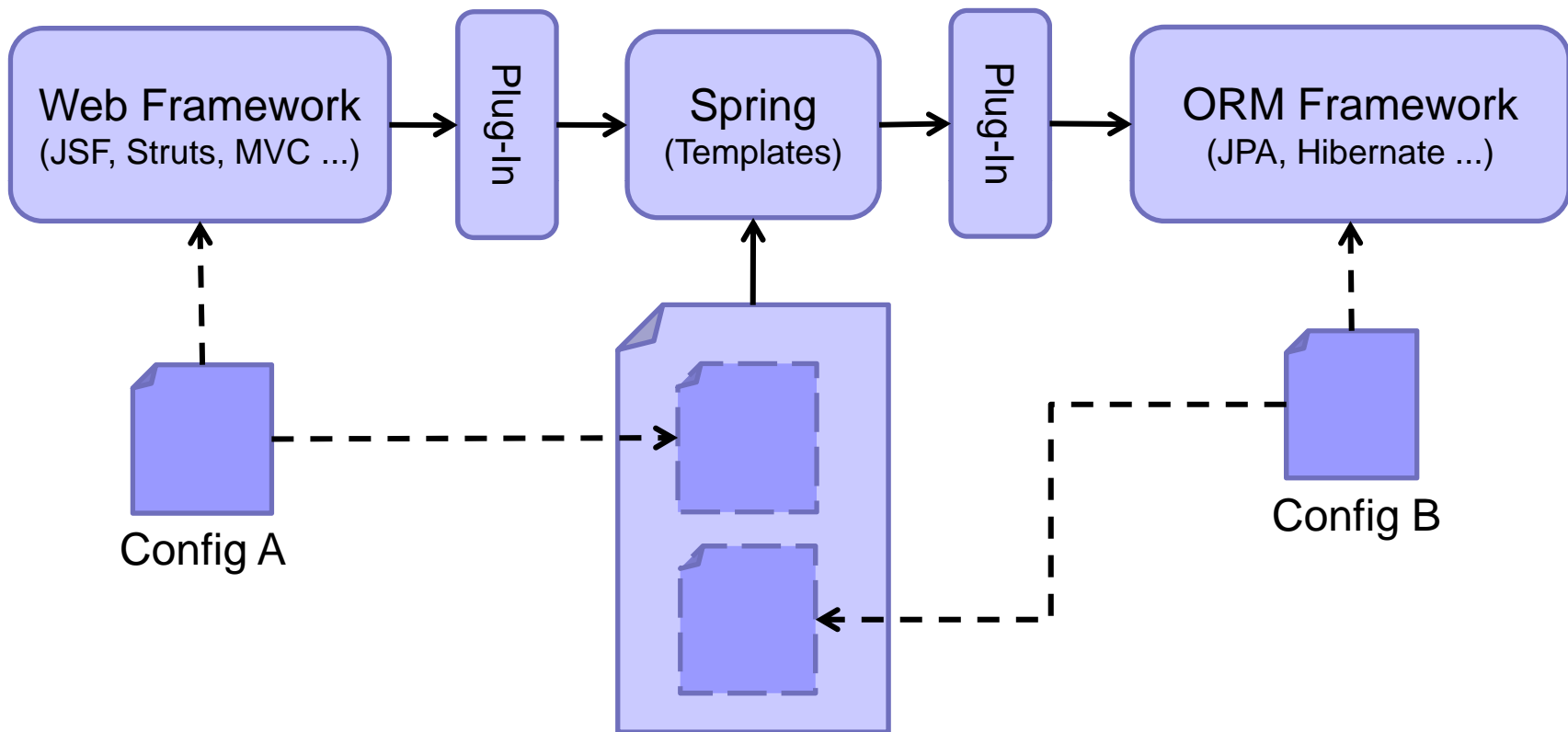
- Spring has three key features
 - The first two features enable the third
- Spring builds beans via dependency injection
 - Which enables your application to be reconfigured on the fly
 - This is very useful if you are adopting Agile Development
- Spring can act as a platform for AOP
 - In a manner that makes adopting AOP much less scary
- Spring works as an integration tool
 - Enabling you to combine multiple open source frameworks
 - Such as Hibernate, JavaServer Faces and Web Services
 - This functionality is built on top of dependency injection



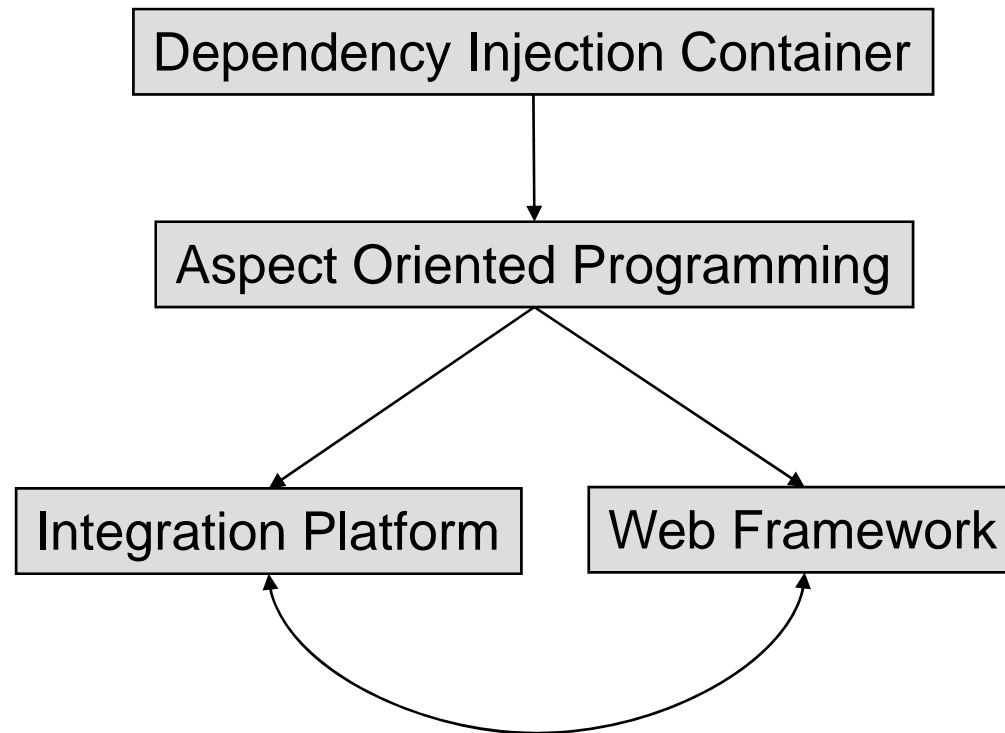
The Spring Framework



Spring as an Integration Platform



Key Features of Spring





Spring as an Integration Platform

- Modern applications combine frameworks
 - Struts for the web tier + Hibernate for ORM
 - A Swing GUI + Web Services to reach the server
 - JSF for the web tier + business logic in EJB's
- Spring simplifies the task of integrating frameworks
 - It provides helper and utility classes for each framework
 - These can be mixed with your code via injection or AOP
- Spring simplifies your use of:
 - Database access libraries (JDBC, iBATIS etc...)
 - ORM frameworks (Hibernate, TopLink, JPA etc...)
 - Web frameworks (Struts, JSF, Tiles, Velocity etc...)
 - Remote objects (RMI, JMS, EJB, Web Services etc...)



Basic Dependency Injection in Spring

Dynamically Wiring
Components Together



Introducing Dependency Injection

- Consider the class below:
 - Q: Does this count as good object oriented design?
 - Q: What problems would this cause for testing / QA?

```
public class Shop {
    public Shop() {
        pricingEngine = new PricingEngine(500,10);
        stockCheckEngine = new StockCheckEngine();
        paymentEngine = new PaymentEngine("www.somewhere.com");
    }
    public boolean makePurchase(String itemNo, int quantity, String cardNo) {
        //details omitted
    }
    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```



Introducing Dependency Injection

- The problem with this class is that it completely encapsulates the creation of its dependencies
 - A 'Shop' is tightly coupled to the same kind of 'PricingEngine', 'PaymentEngine' and 'StockCheckEngine'
- This raises two closely related problems
 - It is impossible to create a unit test for the 'Shop' class
 - The closest we can get is an integration test of four classes
 - We cannot create custom configurations for special needs
 - E.g. during QA or customer demos we might want to have a pricing engine the reads from a comma separated file or local database



Introducing Dependency Injection

- The current best practice is to inject dependencies
 - The class has its dependencies ‘injected’ from the outside
 - Loose coupling is enabled via interfaces and base classes
 - The class is unaware of which implementation it is using
- If class A is dependent on B then we:
 - Extract a base class or interface from B
 - Making room for multiple implementations
 - Refactor A to support injecting dependencies
 - Either via constructor arguments or JavaBean properties

Refactoring Class Dependencies

```
public class PaymentEngine {  
    public boolean authorize(String cardNo, double amount) {  
        return amount < 1000;  
    }  
}
```



```
public interface PaymentEngine {  
    public boolean authorize(String cardNo, double amount);  
}
```

```
public class PaymentEngineStub implements PaymentEngine {  
    public boolean authorize(String cardNo, double amount) {  
        return amount < 1000;  
    }  
}
```

```
public class PaymentEngineWebService implements PaymentEngine {  
    public boolean authorize(String cardNo, double amount) {  
        //implementation omitted  
    }  
}
```



Dependencies Set via Constructors

```
public class Shop {
    public Shop(PricingEngine pricingEngine,
               StockCheckEngine stockCheckEngine,
               PaymentEngine paymentEngine) {
        super();
        this.pricingEngine = pricingEngine;
        this.stockCheckEngine = stockCheckEngine;
        this.paymentEngine = paymentEngine;
    }
    public boolean makePurchase(String itemNo, int quantity, String cardNo) {
        //details omitted
    }
    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```



Dependencies Set via Accessors

```
public class Shop {
    public void setPaymentEngine(PaymentEngine paymentEngine) {
        this.paymentEngine = paymentEngine;
    }
    public void setPricingEngine(PricingEngine pricingEngine) {
        this.pricingEngine = pricingEngine;
    }
    public void setStockCheckEngine(StockCheckEngine stockCheckEngine) {
        this.stockCheckEngine = stockCheckEngine;
    }
    public boolean makePurchase(String itemNo, int quantity, String cardNo) {
        //details omitted
    }
    private PricingEngine pricingEngine;
    private StockCheckEngine stockCheckEngine;
    private PaymentEngine paymentEngine;
}
```



Spring and Dependency Injection

- Spring acts as a universal object factory
 - It reads bean definitions from XML files and instantiates them
 - Definitions can be nested within one another
- Both constructor and property based injection is possible
 - You can stick to either or use a combination

```
public static void main(String[] args) throws Exception {
    BeanFactory factory = new XmlBeanFactory(new FileSystemResource ("config.xml"));
    Shop shop = (Shop)factory.getBean("shopWithMocks");
    if(shop.makePurchase("AB123", 20, "DEF456GHI78")) {
        System.out.println("Purchase Succeeded");
    } else {
        System.out.println("Purchase Failed");
    }
}
```



```
<bean id="paymentImplOne"
  class="demos.v2.PaymentEngineImpl">
  <constructor-arg index="0">
    <value>www.somewhere.com</value>
  </constructor-arg>
  <constructor-arg index="1">
    <value>5000</value>
  </constructor-arg>
</bean>
```

```
<bean id="stockMockOne"
  class="demos.v2.StockCheckEngineMock"/>
<bean id="paymentMockOne"
  class="demos.v2.PaymentEngineMock"/>
```

```
<bean id="pricingMockOne"
  class="demos.v2.PricingEngineMock">
  <constructor-arg index="0">
    <value>500</value>
  </constructor-arg>
  <constructor-arg index="1">
    <value>10</value>
  </constructor-arg>
</bean>
```

Dependency injection
using arguments
to constructors

```
<bean id="shopWithMocks" class="demos.v2.Shop">
  <constructor-arg index="2">
    <ref bean="paymentMockOne"/>
  </constructor-arg>
  <constructor-arg index="1">
    <ref bean="stockMockOne"/>
  </constructor-arg>
  <constructor-arg index="0">
    <ref bean="pricingMockOne"/>
  </constructor-arg>
</bean>
```

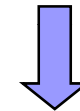
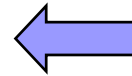


```
<bean id="paymentImplOne"
      class="demos.v3.PaymentEngineImpl">
  <constructor-arg index="0">
    <value>www.somewhere.com</value>
  </constructor-arg>
  <constructor-arg index="1">
    <value>5000</value>
  </constructor-arg>
</bean>


<bean id="stockMockOne"
      class="demos.v3.StockCheckEngineMock"/>
</bean>
<bean id="paymentMockOne"
      class="demos.v3.PaymentEngineMock"/>

<bean id="pricingMockOne"
      class="demos.v3.PricingEngineMock">
  <property name="minimumDiscountAmount">
    <value>500</value>
  </property>
  <property name="percentageDiscount">
    <value>10</value>
  </property>
</bean>
```

Dependency injection
using properties
(e.g. 'setPricingEngine')



```
<bean id="shopWithMocks"
      class="demos.v3.Shop">
  <property name="paymentEngine">
    <ref bean="paymentMockOne"/>
  </property>
  <property name="stockCheckEngine">
    <ref bean="stockMockOne"/>
  </property>
  <property name="pricingEngine">
    <ref bean="pricingMockOne"/>
  </property>
</bean>
```



Dependency injection using
nested bean declarations



```
<bean id="shopWithMocks" class="demos.v4.Shop">
  <property name="paymentEngine">
    <bean class="demos.v4.PaymentEngineMock"/>
  </property>
  <property name="stockCheckEngine">
    <bean class="demos.v4.StockCheckEngineMock"/>
  </property>
  <property name="pricingEngine">
    <bean class="demos.v4.PricingEngineMock">
      <property name="minimumDiscountAmount">
        <value>500</value>
      </property>
      <property name="percentageDiscount">
        <value>10</value>
      </property>
    </bean>
  </property>
</bean>
```



Using the Spring Framework

- A bean factory is the most basic way of using Spring
 - An 'XmlBeanFactory' builds beans based on an XML config file
 - The location of the file is represented by a 'Resource' object
 - There are resource classes to represent the file system, byte arrays, input streams, the classpath, URL's etc...
- Most applications will use an application context
 - This adds support for i18n, event generation and loading files
 - There are several kinds of application context:
 - A 'ClassPathXmlApplicationContext' loads files via the classpath
 - A 'FileSystemXmlApplicationContext' loads files via the file system




The Lifecycle of a Bean

- By default Spring creates singletons
 - Calling `factory.getBean("fred")` always returns the same object
 - Note this applies even if the bean was created indirectly
- It is possible to define a bean as a prototype
 - A new instance is created each time the bean is requested
 - Before V2 you added `'singleton="false"'` to the definition
 - Since V2 you write `'scope="prototype"'` instead
- Spring 2.0 adds additional scopes for Web Applications
 - These are `'request'` , `'session'` and `'global session'`
- You can also specify lifecycle methods for beans
 - Via the `'init-method'` and `'destroy-method'` attributes

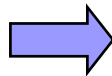


Support for Factory Methods

- Spring allows beans to be created indirectly
 - Via a factory method in the Bean
 - Using the 'factory-method' element
 - Via a factory method in a separate Bean
 - Using the 'factory-bean' and 'factory-method' elements
 - The bean acting as a factory can be configured normally
- The name of the method is arbitrary
 - Normal choices are 'instance' or 'newInstance'
- Parameters are passed via 'constructor-arg' elements
 - The contents are converted into the required type

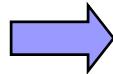


Bean creation via
factory method



```
<bean id="paymentImplOne"  
      class="demos.v5.PaymentEngineImpl"  
      factory-method="instance">  
  <constructor-arg index="0">  
    <value>www.somewhere.com</value>  
  </constructor-arg>  
  <constructor-arg index="1">  
    <value>5000</value>  
  </constructor-arg>  
</bean>
```

Bean creation via
factory object



```
<bean id="paymentImplFactory"  
      class="demos.v6.PaymentEngineFactory"/>  
<bean id="paymentImplOne"  
      class="demos.v6.PaymentEngineImpl"  
      factory-bean="paymentImplFactory"  
      factory-method="instance">  
  <constructor-arg index="0">  
    <value>www.somewhere.com</value>  
  </constructor-arg>  
  <constructor-arg index="1">  
    <value>5000</value>  
  </constructor-arg>  
</bean>
```



Support for Collections

- Spring makes it straightforward to inject collections
 - List, sets and maps and properties can all be declared
- The list and set types do not map directly to collections
 - Spring will create an array, 'java.util.Collection', 'java.util.Set' or 'java.util.List' based on the type of the parameter
 - List members can be 'value', 'ref', 'bean', 'null' or nested lists

Element	Description
list	Both declare a sequence of values, only a list can contain duplicates
set	
map	An implementation of 'java.util.Map' where the keys are strings
props	An implementation of 'java.util.Properties' - both keys and values are strings




```
<bean id="pricingEngine" class="demos.v9.PricingEngineMock">
  <property name="minimumDiscountAmount">
    <value>500</value>
  </property>
  <property name="percentageDiscount">
    <value>10</value>
  </property>
  <property name="prices">
    <list>
      <value>1.20</value>
      <value>3.40</value>
      <value>5.60</value>
    </list>
  </property>
</bean>
<bean id="paymentEngine" class="demos.v9.PaymentEngineMock"/>
<bean id="stockCheckEngine" class="demos.v9.StockCheckEngineMock">
  <property name="testData">
    <map>
      <entry key="ABC123" value="10"/>
      <entry key="DEF456" value="20"/>
      <entry key="GHI789" value="30"/>
    </map>
  </property>
</bean>
<bean id="shopWithMocks" class="demos.v9.Shop" autowire="byName"/>
```

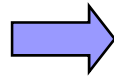


Support for Autowiring

- Autowiring is a powerful but scary feature
 - Spring implicitly works out which bean should be passed as the parameter in a constructor or property
- There are four different types of autowiring
 - 'byName' matches bean names to property names
 - The bean called 'shop' is mapped to the method 'setShop'
 - 'byType' matches bean types to property parameter types
 - If 'setShop' takes a parameter of interface type 'Shop' then the bean that implements that interface will be injected (only one can exist)
 - 'constructor' matches bean types to constructor parameter types
 - 'autodetect' tries to match by constructor first and then by type

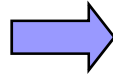


Autowiring
By Type



```
<bean class="demos.v7.PricingEngineMock">
  <property name="minimumDiscountAmount">
    <value>500</value>
  </property>
  <property name="percentageDiscount">
    <value>10</value>
  </property>
</bean>
<bean class="demos.v7.PaymentEngineMock"/>
<bean class="demos.v7.StockCheckEngineMock"/>
<bean id="shopWithMocks" class="demos.v7.Shop" autowire="byType"/>
```

Autowiring
By Name



```
<bean id="pricingEngine" class="demos.v8.PricingEngineMock">
  <property name="minimumDiscountAmount">
    <value>500</value>
  </property>
  <property name="percentageDiscount">
    <value>10</value>
  </property>
</bean>
<bean id="paymentEngine" class="demos.v8.PaymentEngineMock"/>
<bean id="stockCheckEngine" class="demos.v8.StockCheckEngineMock"/>
<bean id="shopWithMocks" class="demos.v8.Shop" autowire="byName"/>
```



Using Inheritance in Definitions

- Autowiring is one way of minimizing configuration
 - Another way is declaring parent definitions
- One definition can be the specialization of another
 - The base definition doesn't need any extra markup
 - However you will probably want to declare it with 'abstract=true' to prevent Spring generating instances
 - The derived definition uses 'parent=baseBean'
 - It can add its own properties and override inherited ones
 - Clients instantiate derived definitions as normal
 - The use of inheritance does not affect client code



```
<!-- The parent bean declaration -->
```

```
<bean id="shopWithMocks" abstract="true" class="demos.v10.Shop">  
  <property name="paymentEngine"><ref bean="paymentEngine" /></property>  
  <property name="stockCheckEngine"><ref bean="stockCheckEngine" /></property>  
  <property name="pricingEngine"><ref bean="pricingEngine" /></property>  
</bean>
```

```
<!-- Three derived declarations -->
```

```
<bean id="testShop1" parent="shopWithMocks">  
  <property name="shopName">  
    <value>Test Shop No 1</value>  
  </property>  
</bean>  
<bean id="testShop2" parent="shopWithMocks" >  
  <property name="shopName">  
    <value>Test Shop No 2</value>  
  </property>  
</bean>  
<bean id="testShop3" parent="shopWithMocks" >  
  <property name="shopName">  
    <value>Test Shop No 3</value>  
  </property>  
</bean>
```



Extra Dependency Injection Features

Power Features in Spring




Additional Dependency Injection

- Spring offers advanced dependency injection features:
 - Derived beans can override inherited settings
 - Method definitions in beans can be replaced
 - Property editors can be used to pass user defined types into setter methods and constructors
 - Information can be read from properties files
 - Beans can be made aware of the Spring framework
 - By listening for events or having context objects injected
 - The dependency injection process can be customized
 - By creating post-processors for beans and bean factories
 - Beans can be written using scripting languages



Overriding Inherited Properties

- We have already seen inheritance in bean definitions
 - This is normally done to reuse shared properties
- There are two ways of customizing inheritance
 - A bean can override inherited properties
 - In the same manner as method overriding in Java
 - You can also override ‘init-method’ and ‘destroy-method’
 - A bean declaration does not have to reference a class
 - It can simply be used to hold common properties
 - The declaration always needs to have “abstract=true”
 - The classes whose bean declarations inherit from such a declaration do not need to share a common base class



```
<bean id="sharedProperties" abstract="true">
  <property name="number"><value>10</value></property>
  <property name="street"><value>Arcatia Road</value></property>
  <property name="postcode"><value>BT37ABC</value></property>
</bean>

<bean id="baseEmployee" parent="sharedProperties" abstract="true"
  class="demos.overriding.Employee">
  <property name="title"><value>GeneralStaff</value></property>
  <property name="salary"><value>20000.0</value></property>
</bean>

<bean id="employee" parent="baseEmployee">
  <property name="title"><value>SoftwareDeveloper</value></property>
  <property name="fullName"><value>Dave Jones</value></property>
</bean>

<bean id="company" parent="sharedProperties" class="demos.overriding.Company">
  <property name="vatRegistration"><value>ABC123</value></property>
  <property name="taxes"><value>4500.25</value></property>
</bean>
```



Replacing Bean Method Definitions

- Spring can replace the definitions of methods
 - A method invocation can be redirected away from the intended bean and towards a different one
- This feature is enabled via proxies
 - Spring creates a new class that implements the same interface as your bean and returns an instance
 - Proxies are the key to implementing AOP
- In order to replace a method:
 - Place a 'replaced-method' tag in the bean declaration
 - Create a class that implements the 'MethodReplacer' interface
 - The new functionality is placed in the 'reimplement' method



Replacing Bean Method Definitions

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

  <bean id="addReplacer" class="demos.altering.methods.MathsReplacer"/>

  <bean id="maths1" class="demos.altering.methods.MathsImpl"/>

  <bean id="maths2" class="demos.altering.methods.MathsImpl">
    <replaced-method name="addNumbers" replacer="addReplacer">
      <arg-type>int</arg-type>
      <arg-type>int</arg-type>
    </replaced-method>
  </bean>
</beans>
```



Replacing Bean Method Definitions

```
public interface Maths {  
    int addNumbers(int no1, int no2);  
}
```

```
public class MathsImpl implements Maths {  
    public int addNumbers(int no1, int no2) {  
        return no1 + no2;  
    }  
}
```

```
public class MathsReplacer implements MethodReplacer {  
    public Object reimplement(Object target, Method method,  
        Object[] params) throws Throwable {  
        int param1 = Integer.parseInt(params[0].toString());  
        int param2 = Integer.parseInt(params[1].toString());  
        return param1 * param2;  
    }  
}
```



Using JavaBean Property Editors


- Property Editors are part of the JavaBeans specification
 - They allow user-defined types to be used in setter methods
 - This feature was intended for GUI tools and is rarely used
- A Property Editor class extends 'PropertyEditorSupport'
 - The 'setAsText' method takes a string, parses it and uses the extracted data to populate a user-defined type
 - This UDT instance is then passed back via 'setValue'
- In order to register Property Editors with Spring
 - Declare an instance of 'CustomEditorConfigurer' in the bean file
 - Populate the 'customEditors' property with a map
 - Where the key is the UDT name and the value is a bean class



```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
      <map>
        <entry key="demos.editors.Address">
          <bean id="addressEditor" class="demos.editors.AddressPropertyEditor"/>
        </entry>
      </map>
    </property>
  </bean>

  <bean id="employee" class="demos.editors.Employee">
    <property name="fullName">
      <value>Dave Smith</value>
    </property>
    <property name="address">
      <value>10#Arcatia Road#Belfast#BT37 AB7</value>
    </property>
  </bean>
</beans>
```




```
public class AddressPropertyEditor extends PropertyEditorSupport {
    @Override
    public String getAsText() {
        return getValue().toString();
    }
    @Override
    public void setAsText(String input) throws IllegalArgumentException {
        Pattern pattern = Pattern.compile("[0-9]{2}#[A-Za-z ]+#[A-Za-z ]+#[A-Z0-9 ]+");
        Matcher matcher = pattern.matcher(input.trim());
        if(matcher.matches()) {
            int tmpNumber = Integer.parseInt(matcher.group(1));
            String tmpStreet = matcher.group(2);
            String tmpCity = matcher.group(3);
            String tmpPostcode = matcher.group(4);
            Address address = new Address(tmpNumber,tmpStreet,tmpCity,tmpPostcode);
            setValue(address);
        }
    }
}
```



Reading Info From Property Files

- You can use property files in a Spring based system:
 - Perhaps because they were part of the pre-Spring architecture
 - Or in order to allow non-developers to change system settings
- There are two ways of including property files
 - By declaring an instance of 'PropertyPlaceholderConfigurer'
 - By using the 'property-placeholder' schema extension
 - The value is a comma separated list of file paths
- Properties are accessed via the '\${name}' syntax
 - The same syntax as ANT files and JSP EL



```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <!-- <property name="location" value="app.properties" /> -->
  <property name="locations">
    <list>
      <value>demos/properties/app1.properties</value>
      <value>demos/properties/app2.properties</value>
      <value>demos/properties/app3.properties</value>
      <value>demos/properties/app4.properties</value>
    </list>
  </property>
</bean>

<bean name="employee" class="demos.properties.Employee">
  <property name="name" value="${employee.name}"/>
  <property name="age" value="${employee.age}"/>
  <property name="department" value="${employee.dept}"/>
  <property name="address" value="${employee.address}"/>
  <property name="salary" value="${employee.salary}"/>
  <property name="married" value="${employee.isMarried}"/>
  <property name="holidaysRemaining" value="${employee.holidaysRemaining}"/>
  <property name="sickDaysThisYear" value="${employee.sickDaysThisYear}"/>
</bean>
```



```
<?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
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-2.5.xsd">


  <context:property-placeholder location="demos/properties/app1.properties,
    demos/properties/app2.properties,
    demos/properties/app3.properties"/>

  <bean name="employee" class="demos.properties.Employee">
    <property name="name" value="{employee.name}"/>
    <property name="age" value="{employee.age}"/>
    <property name="department" value="{employee.dept}"/>
    <property name="address" value="{employee.address}"/>
    <property name="salary" value="{employee.salary}"/>
    <property name="married" value="{employee.isMarried}"/>
    <property name="holidaysRemaining" value="{employee.holidaysRemaining}"/>
    <property name="sickDaysThisYear" value="{employee.sickDaysThisYear}"/>
  </bean>
</beans>
```




Creating Spring-Aware Beans

- All the Beans we have built so far are POJO's
 - They have no knowledge of the container that is hosting them
- It is possible to make beans aware of the container
 - By implementing callback interfaces provided by Spring
- There are three callback interfaces:
 - 'BeanNameAware' enables a bean to discover its name
 - 'BeanFactoryAware' enables a bean to access the factory
 - E.g. to discover if a bean exists and what scope it is in
 - 'ApplicationContextAware' provides access to the context
 - The bean can explore the full framework configuration



```
public class MyBeanOne implements BeanNameAware {
    public void setBeanName(String name) {
        this.name = name;
    }
    public void sayName() {
        System.out.println("My name is: " + name);
    }
    private String name;
}
```

```
public class MyBeanThree implements ApplicationContextAware {
    public void setApplicationContext(ApplicationContext context) throws BeansException {
        this.context = context;
    }
    public void listBeans() {
        System.out.println("Deployed beans are: ");
        for(String name : context.getBeanDefinitionNames()) {
            System.out.printf("\t%s of type %s\n", name, context.getType(name).getName());
        }
    }
    private ApplicationContext context;
}
```




```
public class MyBeanTwo implements BeanFactoryAware {
    public void setBeanFactory(BeanFactory factory) throws BeansException {
        this.factory = factory;
    }
    public void printBeanDetails(String beanName) {
        if(factory.containsBean(beanName)) {
            System.out.printf("%s is an instance of %s",
                beanName, factory.getType(beanName).getName());
            if(factory.isPrototype(beanName)) {
                System.out.println(" in prototype scope");
            } else {
                System.out.println(" in singleton scope");
            }
        }
    }
    private BeanFactory factory;
}
```



Customizing Dependency Injection

- The Spring framework can be customized
 - By writing post processor components that perform extra work
- There are two types of post processor
 - A bean post processor can modify a bean both before and after its properties are set and init method called
 - A bean factory post processor can modify the tree of bean declarations created by reading from the XML file
 - This could be used to add extra beans into the scope of the factory
 - E.g. by finding all the classes that implement a particular interface
- Registering post processors with Spring is easy
 - Any declared bean that implements 'BeanPostProcessor' or 'BeanFactoryPostProcessor' will be found and registered



```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {  
    public void postProcessBeanFactory(ConfigurableListableBeanFactory factory)  
        throws BeansException {  
        System.out.println("MyBeanFactoryPostProcessor.postProcessBeanFactory called!");  
    }  
}
```

```
public class MyBeanPostProcessor implements BeanPostProcessor {  
    public Object postProcessAfterInitialization(Object bean, String name) throws BeansException {  
        System.out.println("After init called for " + name + " of type " + bean.getClass().getName());  
        return bean;  
    }  
    public Object postProcessBeforeInitialization(Object bean, String name) throws BeansException {  
        System.out.println("Before init called for " + name + " of type " + bean.getClass().getName());  
        return bean;  
    }  
}
```



Creating Beans Via Scripts

- Scripting is becoming a key part of the Java platform
 - As the language itself becomes more and more complex
- Spring supports beans implemented using scripts
 - At present 'JRuby', 'Groovy' and 'BeanShell' are supported
 - The details of configuring and using each are slightly different
 - The scripts can be placed in two locations
 - Either in separate files or inline in the Spring configuration file
- There is one major advantage to scripted beans
 - The framework can reload scripts when the file is modified
 - This creates 'refreshable beans' whose implementation can be changed without needing to restart the application