Annotation Based Configuration

- The JDK 1.5 introduced annotations as a way of providing meta-tags
  - Meta-tags tell the compiler and other programs how to interpret program elements (methods, declarations, parameters, etc)
- Spring allows annotations to be used instead of XML
  - You can mix the two styles

- Three Styles of providing the Container Metadata
  - **XML** based (traditional format)
  - **Annotation** based (introduced in Spring 2.5)
  - **Java** based (introduced in Spring 3.0)

Autowiring by XML (Review)

- Spring attempts to match all properties of a bean with beans of the same name
- Properties that have no match will remain unwired
  - The order of beans in your .xml does not matter

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
   private ShippingChargeCalculator shippingCalculator;
   private String companyName;
```

```
<bean id="shippingCalculator" class="nvz. ShippingChargeCalculator" />

<bean id="invoiceGenerator" class="nvz.services.InvoiceGeneratorImpl"
      autowire="byName">
        <property name="companyName" value="ZBooks" />
</bean>
```

Autowiring by Annotation

- Use @Autowired on a property, a set method, or a constructor

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
    @Autowired
    private ShippingChargeCalculator shippingCalculator;
    private String companyName;
```

```
<bean id="shippingCalculator" class="nvz. ShippingChargeCalculator" />

<bean id="invoiceGenerator" class="nvz.services.InvoiceGeneratorImpl">
        <property name="companyName" value="ZBooks" />
</bean>
```

- Spring must find exactly one bean that is a match
  - An exception occurs if there is no match
  - An exception occurs if there is more than one match

Autowiring (Matching by Type)

- @Autowired matching is done by type.

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
   @Autowired
   private ShippingChargeCalculator shippingCalculator;
   private String companyName;
```

```
<bean id="californiaShippingCalculator"
      class="nvz.services.ShippingChargeCalculatorSimpleImpl" />

<bean id="invoiceGenerator" class="nvz.services.InvoiceGeneratorImpl">
      <property name="companyName" value="ZBooks" />
</bean>
```

- In the above XML configuration, there is one bean with a type of ShippingChargeCalculator
  - The californiaShippingCalculator bean will be injected as it is a type match

Autowiring by Annotation

- By default, Spring doesn't look for annotations in your code.
- Turn it on by adding the following XML

```
<context:annotation-config />
```

- The **context:annotation-config** tag tells Spring you want to do annotation based wiring and it will then scan your classes looking for annotations

Qualifying Ambiguous Dependencies

- The **@Qualifier** tag may be used to help Spring make a unique match

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
    @Autowired
    @Qualifier("californiaShippingCalculator")
    private ShippingChargeCalculator shippingCalculator;
    private String companyName;
```

```
<bean id="californiaShippingCalculator"
        class="nvz.services.ShippingChargeCalculatorSimpleImpl" />

<bean id="nevadaShippingCalculator"
        class="nvz.services.ShippingChargeCalculatorSimpleImpl" />

<bean id="invoiceGenerator" class="nvz.services.InvoiceGeneratorImpl">
        <property name="companyName" value="ZBooks" />
</bean>
```

- In the above XML configuration, there are two ShippingChargeCalculator beans that match by type
  - Ambiguous if not qualified

Qualifying Ambiguous Dependencies

- The **@Qualifier** tag can also match based on values specified with **<qualifier>**

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
   @Autowired
   @Qualifier("default_shippingCalculator")
   private ShippingChargeCalculator shippingCalculator;
   private String companyName;
```

```
<bean id="californiaShippingCalculator"
      class="nvz.services.ShippingChargeCalculatorSimpleImpl" >
    <qualifier value="default_shippingCalculator" />
</bean>

<bean id="nevadaShippingCalculator"
      class="nvz.services.ShippingChargeCalculatorSimpleImpl" />

<bean id="invoiceGenerator" class="nvz.services.InvoiceGeneratorImpl">
      <property name="companyName" value="ZBooks" />
</bean>
```

- In the above XML configuration, there are two ShippingChargeCalculator beans that can match by type
  - A **default_shippingCalculator** must be found

Qualifying Ambiguous Dependencies

- The @Qualifier could also be annotated on a class to narrow the matches

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
   @Autowired
   @Qualifier("default_shippingCalculator")
   private ShippingChargeCalculator shippingCalculator;
   private String companyName;
```

```
@Qualifier("default_shippingCalculator")
public class ShippingChargeCalculatorSimpleImpl
            implements ShippingChargeCalculator
{   ...   }
```

```
public class IntlShippingChargeCalcImpl
            implements ShippingChargeCalculator
{   ...   }
```

```
<bean id="californiaShippingCalculator"
      class="nvz.services.ShippingChargeCalculatorSimpleImpl" />

<bean id="worldShippingCalculator"
      class="nvz.services.IntlShippingChargeCalcImpl" />
```

- Which of the two beans in the XML configuration is a match for the ShippingChargeCalculator?
  - In this case, the **class** marked as default_shippingCalculator

Autowiring Matching

- Matching will be done in the following order (criteria for elimination):
  1. Type
  2. Qualifier
  3. Name

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
   @Autowired
   private ShippingChargeCalculator shippingCalculator;
   private String companyName;
```

```
<bean id="californiaShippingCalculator"
      class="nvz.services.ShippingChargeCalculatorSimpleImpl" />

<bean id="shippingCalculator"
      class="nvz.services.ShippingChargeCalculatorSimpleImpl" />

<bean id="invoiceGenerator" class="nvz.services.InvoiceGeneratorImpl">
      <property name="companyName" value="ZBooks" />
```

- In the above XML configuration, there are two ShippingChargeCalculator beans
  - californiaShippingCalculator and shippingCalculator
  - The bean with the name shippingCalculator is the match (by name)

Optional Autowiring

- By default, autowiring fails if no match is found
- The behavior can be changed
  - Set the **required** attribute to **false**
  - If no match is found, the property will have value null

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
  @Autowired(required=false)
  private ShippingChargeCalculator shippingCalculator;
```

@Required

- If you are not using @Autowired but still want to specify that a property must be set, use **@Required**
- May only be applied on set methods

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
  private ShippingChargeCalculator shippingCalculator;

  @Required
  public void  setShippingCalculator(ShippingChargeCalculator shippingCalc) {
    this.shippingCalculator = shippingCalc;
  }
```

- You'll get an exception when you attempt to run the program if the shippingCalculator property is never set:

```
Caused by: org.springframework.beans.factory.BeanInitializationException: Property 'shippingCalculator' is required for bean
 'invoiceGenerator' at
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor.postProcessPropertyValues(RequiredAnn
otationBeanPostProcessor.java:151)
    at
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.populateBean(AbstractAutowireCapableBeanF
actory.java:1120)
    at
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBean
Factory.java:522)
```

Annotating Simple Properties

- How can we simplify the XML when there are simple properties?
  - Use **@Value**

```
<bean id="invoiceGenerator"
      class="nvz.services.InvoiceGeneratorImpl">
  <property name="shippingCalculator" ref="shippingCalculator"/>
  <property name="companyName" value="${invoice.companyName}" />
  <property name="companyId" value="${invoice.companyId}" />
  <property name="salesTax" value="${invoice.salesTax}" />
</bean>
```

Reduce the XML for the **invoiceGenerator** bean

```
<bean id="invoiceGenerator"
      class="nvz.services.InvoiceGeneratorImpl"/>
```

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
  @Value("${invoice.companyName}")
  private String companyName;
  @Value("${invoice.companyId}")
  private int companyId;
  @Value("${invoice.salesTax}")
  private double salesTax;
  @Autowired
  @Qualifier("default_shippingCalculator")
  private ShippingChargeCalculator shippingCalculator;
```

@Value

- Note that you could also use constants in @Value

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
  @Value("zBooks")
  private String companyName;
```

- But it would be better just to hardcode the data
  - Approach on previous slide is better because no recompilation is necessary to make changes (and don't have to dig through code)

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
  private String companyName = "zBooks";
```

Spring Annotations vs. Standards

- In some cases you'll find you have a choice of annotations to use
  - **Spring based** annotations
    - Found in the Spring packages
  - **Standards based** annotations
    - Specified by the committee that sets Java standards
    - JSRs (Java Specification Request) made to the Java Community Process that become official

Spring Annotations vs. Standards

- **Spring based** annotations
  - From `org.springframework.beans.factory.annotation`
    - @Autowired
    - @Qualifier
- **Standards based** annotations
  - JSR-330 Annotations (may have to add JSR 330 .jar file to your classpath)
    - @Inject similar to @Autowired annotation
    - @Qualifier used to create custom qualifiers
    - @Named specifies bean name
  - JSR-250 Annotation
    - @Resource declares a reference to a resource
      use a bean id (name) – standard Java annotation
      `javax.annotation` package

Standards Based Annotations

- **Use @Inject in place of @Autowired**
- **Use @Named in place of @Qualifier**
  - **Different than @Qualifier in that it specifically provides a bean id (name)**

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
  @Inject
  @Named("californiaShippingCalculator")
  private ShippingChargeCalculator shippingCalculator;
```

- JSR 330 Limitations
  - No required attribute
  - No @Required
  - No @Value

JSR 250 @Resource

- **JSR 250 (Common Annotations for the Java Platform)**
  - **These annotations are part of standard Java (as of 1.6) and are not Spring-specific**
  - **Can be used on properties or setter methods**
  - **Provide a bean id (name), or use type matching if a name is not provided**

```
public class InvoiceGeneratorImpl implements InvoiceGenerator {
  @Resource(name="californiaShippingCalculator")
  private ShippingChargeCalculator shippingCalculator;
```

Meta Data Precedence

- **Annotations are processed before XML**
- **Therefore if data is specified both in annotation and XML, the XML wins out**

Automatically Discovering Beans

- We've seen bean definitions done in XML
- Spring annotations can also allow bean definitions to be inferred from annotations
- To enable this feature, use the **component-scan** tag in your XML
  - Replaces the **annotation-config** tag
  - Scanning is not done if tag is not present
- Auto-inference through annotation lets us reduce the XML even more

```
<context:component-scan base-package="nvz.services"/>
```

- Need to provide the base package
  - Spring only scans packages specified
  - All classes in the package and any subpackages will be scanned for annotations

Annotations for Auto Discovery

- **@Component**
  - General purpose specifier – any bean type

  Or you can be more specific about the type of bean you are creating:

- **@Service**
  - A bean in the service (business) layer
- **@Repository**
  - A bean in the DAO layer
- **@Controller**
  - A bean that defines a Spring MVC Controller (web layer)

Automatically Discovering Beans

➢ @Component

- Used on a class definition
- Tells Spring to create a bean that is an instance of the class

Replace the XML bean definition

```
<bean id="californiaShippingCalculator"
      class="nvz.services.ShippingChargeCalculatorSimpleImpl" />
```

With an annotation that Spring will discover and create a bean with the same name specified (if a name is not provided the bean name will be the class name)

```
@Component("californiaShippingCalculator")
public class ShippingChargeCalculatorSimpleImpl
                implements ShippingChargeCalculator
{
    public double shippingCharge(Order order) {
       return order.getAmount() * .10;
    }
}
```

Automatically Discovering Beans

- **Use @Service, @Repository, @Controller to be more specific**
  - In the future Spring may use this more specific information to infer additional services required for that particular layer

```
@Service("californiaShippingCalculator")
public class ShippingChargeCalculatorSimpleImpl
                  implements ShippingChargeCalculator
{
    public double shippingCharge(Order order) {
       return order.getAmount() * .10;
    }
}
```

```
@Repository("orderDAO")
public class OrderDAOImpl implements OrderDAO
{
    public double persistOrder(Order order) {
       ...
    }
}
```

Filtering Component Scans

- **To narrow component scanning use**
  - **<context:include-filter>**
  - **<context:exclude-filter>**

```
<context:component-scan base-package="nvz.services">
    <context:include-filter type="regex"
            expression="nvz.services.*Calculator*"/>
    <context:exclude-filter type="assignable"
           expression="nvz.services.ShippingChargeCalculator"/>
</context:component-scan>
```

Types:

- **regex**

  regular expression specifies inclusion/exclusion
- **assignable**

  Spring will automatically register (or exclude) any classes that are assignable to a particular type

For additional types, see the course textbook

A Limitation of Auto Bean Discovery

- **Only one style of the bean can be created using auto discovery**

```
@Service("californiaShippingCalculator")
public class ShippingChargeCalculatorSimpleImpl
                implements ShippingChargeCalculator
{
    @Value("${shipping.shipperName}")
    private String shipper;

    public double shippingCharge(Order order) {
       return order.getAmount() * .10;
    }
}
```

- With XML based configuration we can create multiple beans from the ShippingChargeCalculatorSimpleImpl class
  - And use different properties for each bean

```
<bean id="californiaShippingCalculator"
      class="nvz.services.ShippingChargeCalculatorSimpleImpl" >
    <property name="shipper" value="UPS" />
</bean>

<bean id="intlShippingCalculator"
      class="nvz.services.ShippingChargeCalculatorSimpleImpl" >
    <property name="shipper" value="DHL" />
</bean>
```

Java Based Configuration

- Spring 3.0 introduced another non-XML approach that can provide most of the container meta-data
  - Use pure Java code
  - Still minimally requires an XML file with the **<context:component-scan />** tag

➢ Three Styles of providing the Container Metadata
  - ➢ **XML** based (traditional format)
  - ➢ **Annotation** based (introduced in Spring 2.5)
  - ➢ **Java** based (introduced in Spring 3.0)

A Configuration Class

- **A Java class in which you declare your beans**
  - **Specified by @Configuration**
  - **Methods of the class will be used to declare bean definitions**

```
package nvz.configurations;
import org.springframework.context.annotation.*;

@Configuration
public class BeanContext {

  // Bean Definitions Go Here

}
```

A Configuration Class

- **@Bean is placed on methods to declare bean definitions**

```
package nvz.configurations;

import nvz.services.*;
import org.springframework.context.annotation.*;

@Configuration
public class BeanContext {
  @Bean
  public ShippingChargeCalculator californiaShippingCalculator() {
    ShippingChargeCalculator shipCalc =
                  new ShippingChargeCalculatorSimpleImpl();
    shipCalc.setShipper("UPS");
    return shipCalc;
  }

  @Bean
  public ShippingChargeCalculator intlShippingChargeCalc() {
    ShippingChargeCalculator shipCalc =
                  new ShippingChargeCalculatorSimpleImpl();
    shipCalc.setShipper("DHL");
    return shipCalc;
  }
}
```

Wiring in References

- **In XML we used a ref tag to wire in references to other beans**
- **In Java Configurations, call a function**

Wire a ShippingCalculator into the InvoiceGenerator

```
package nvz.configurations;

import nvz.services.*;
import org.springframework.context.annotation.*;

@Configuration
public class BeanContext {
  @Bean
  public ShippingChargeCalculator intlShippingChargeCalc () {
    ShippingChargeCalculator shipCalc =
                  new ShippingChargeCalculatorSimpleImpl();
    shipCalc.setShipper("UPS");
    return shipCalc;
  }

  @Bean
  public InvoiceGenerator invoiceGenerator () {
    InvoiceGeneratorImpl invGen = new InvoiceGeneratorImpl();
    invGen.setShippingCalculator(intlShippingChargeCalc());
  }
}
```

A Configuration Class

- **Convert our invoiceGenerator bean with properties from XML to Java**
  - Wire in an Environment object to extract the properties

```
@Configuration
@PropertySource({"classpath:invoice.properties",
                 "classpath:database.properties"})
public class BeanContext {
  @Autowired
  private Environment env;    // needed to access properties

  @Bean
  public ShippingChargeCalculator intlShippingChargeCalc() {
    ShippingChargeCalculator shipCalc =
                    new ShippingChargeCalculatorSimpleImpl();
    shipCalc.setShipper("DHL");
    return shipCalc;
  }

  @Bean
  public InvoiceGenerator invoiceGenerator () {
    InvoiceGeneratorImpl invGen = new InvoiceGeneratorImpl();
    invGen.setShippingCalculator(intlShippingChargeCalc());

    String companyName = env.getProperty("invoice.companyName");
    invGen.setCompanyName(companyName);

    int companyId = Integer.parseInt(env.getProperty("invoice.companyId"));
    invGen.setCompanyId(companyId);

    double salesTax = Double.parseDouble(env.getProperty("invoice.salesTax"));
    invGen.setSalesTax(salesTax);

    return invGen;
  }
}
```

Minimal XML

- **Some XML is still required**
  - **Minimally the component-scan tag is needed to scan for @Configuration**
  - **You can mix all 3 meta-data approaches**

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

<context:component-scan
          base-package="nvz.configurations"/>

</beans>
```

Bootstrapping Using Java

- **Instead of using XML you can bootstrap the container using your Configuration classes**
  - **Use the AnnotationConfigApplicationContext class**
  - **Then an XML file is not required**

```
package nvz.client;

import nvz.configurations.BeanContext;
import nvz.domain.*;
import nvz.services.*;

import org.springframework.context.*;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class InvoiceApp {
  public static void main(String args[]) {
    ApplicationContext container = new
            AnnotationConfigApplicationContext(BeanContext.class);

    InvoiceGenerator invGenerator = (InvoiceGenerator)
                container.getBean("invoiceGenerator");
    Order order;

    order = new Order("GSX-56789");
    order.setAmount(20.0);
    invGenerator.produceInvoice(order);
  }
}
```

A Casting Note

- **A newer version of the getBean function allows you to remove the cast**

Object **getBean**(String name, Class requiredType)

- **Old Style**

```
public class InvoiceApp {
  public static void main(String args[]) {
    ApplicationContext container = new
            AnnotationConfigApplicationContext(BeanContext.class);

    InvoiceGenerator invGenerator = (InvoiceGenerator)
                    container.getBean("invoiceGenerator");
  }
}
```

- **New Style**

```
public class InvoiceApp {
  public static void main(String args[]) {
    ApplicationContext container = new
            AnnotationConfigApplicationContext(BeanContext.class);

    InvoiceGenerator invGenerator =
      container.getBean("invoiceGenerator", InvoiceGenerator.class);
  }
}
```

Lazy Init Configuration

- **Use @Lazy if you want your beans instantiated on request (not on startup)**

```
package nvz.configurations;

import nvz.services.*;
import org.springframework.context.annotation.*;

@Configuration
@Lazy(value = true)
public class BeanContext {
  @Bean
  public ShippingChargeCalculator californiaShippingCalculator() {
    ShippingChargeCalculator shipCalc =
                  new ShippingChargeCalculatorSimpleImpl();
    shipCalc.setShipper("UPS");
    return shipCalc;
  }
}
```

Scope Configuration

- **Use @Scope to change your bean from singleton to prototype scope**

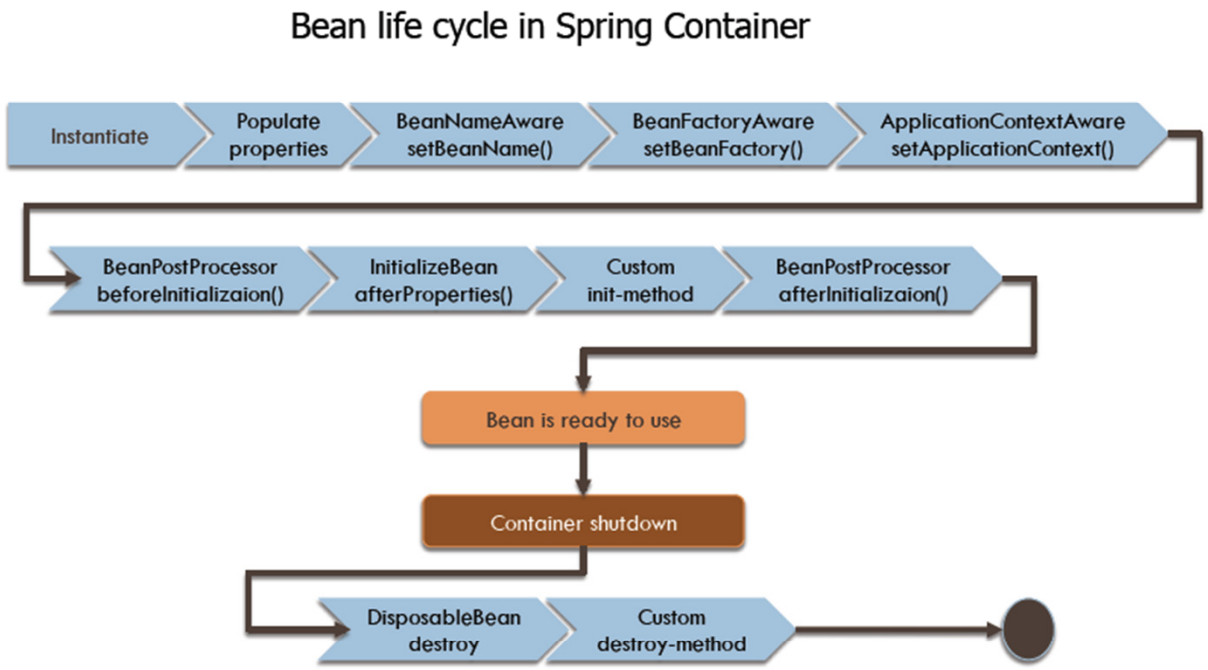
```
package nvz.configurations;

import nvz.services.*;
import org.springframework.context.annotation.*;

@Configuration
@Scope("prototype")
public class BeanContext {
  @Bean
  public ShippingChargeCalculator californiaShippingCalculator() {
    ShippingChargeCalculator shipCalc =
                  new ShippingChargeCalculatorSimpleImpl();
    shipCalc.setShipper("UPS");
    return shipCalc;
  }
}
```

Managing the Bean Lifecycle

- **You may want to take certain actions after a bean has been created or before it is destroyed**
  - **Spring provides such capabilities**



Customizing Init and Destroy Logic

- **A bean intended to store data in a database**
  - **Create a connection to the DB on creation**
  - **Release connection before destruction**

```
public class InvoiceDAOImpl implements InvoiceDAO {
   private DatabaseConnection connection;

   public void initializeDbConn() {
      connection = DatabaseConnection.getInstance();
   }

   public void persistInvoice(Invoice inv) {
      // Save the invoice in the database
   }

   public void releaseDbConn() {
      connection.releaseConnection();
   }
}
```

- We'd like the Spring container to call initializeDbConn() after the bean is instantiated and its properties are set
- Call releaseDbConn() before the container destroys the bean

Using XML to Customize Lifecycle

- **init-method attribute tells Spring to call the method to do initialization**
- **destroy-method attribute tells Spring to call the method as part of shutting down the bean**

```
<bean id="invoiceDao" class="nvz.daos.InvoiceDAOImpl"
      init-method="initializeDbConn"
      destroy-method="releaseDbConn"  />
```

A Shutdown Problem

- **How does the Spring container know when your application ends?**
  - **It doesn't**
  - **More code is necessary if you want to shutdown the Spring container gracefully and have it call your destroy methods**
  - **The JVM provides a way to do this through "shutdown" hooks – or to put it another way, provide a callback method on shutdown**
    - **Register with JVM**
    - **When JVM shuts down it will call all registered shutdown**

Registering a Shutdown Hook

- **Use a reference to AbstractApplicationContext**
  - **It has a method that lets you register the shutdown hook**

```
public class InvoiceApp {
  public static void main(String args[]) {
    AbstractApplicationContext container =
           new ClassPathXmlApplicationContext("application.xml");
    Order order;

    container.registerShutdownHook();

    order = new Order("GSX-56789");
    order.setAmount(20.0);
    invGenerator.produceInvoice(order);
  }
}
```

An Alternative to Shutdown Hooks

- **Alternatively, you can shutdown the Spring container yourself which would allow it to shutdown gracefully**
  - **Use the close() method of AbstractApplicationContext**

```
public class InvoiceApp {
  public static void main(String args[]) {
    AbstractApplicationContext container =
            new ClassPathXmlApplicationContext("application.xml");
    Order order;

    order = new Order("GSX-56789");
    order.setAmount(20.0);
    invGenerator.produceInvoice(order);
    container.close();   // Spring container shutdown here
  }
}
```

Singleton vs. Prototype Beans

- **Lifecycles of prototype-scoped beans and singleton-scoped beans are the same except for destroy methods**
  - **The Spring container will not call the destroy method of prototype-scoped beans**
  - **The object that fetches the prototype-scoped bean from the ApplicationContext is responsible for calling any destroy methods**

Specifying Default Lifecycle Methods

- Apply default attributes to the root **<beans>** element instead of individual beans
  - **default-init-method attribute**
  - **default-destroy-method attribute**

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:c="http://www.springframework.org/schema/c"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                  http://www.springframework.org/schema/beans/spring-beans.xsd
                   http://www.springframework.org/schema/context
                http://www.springframework.org/schema/context/spring-context.xsd
                http://www.springframework.org/schema/util
                http://www.springframework.org/schema/util/spring-util.xsd"
      default-init-method="initDbConn"
      default-destroy-method="releaseDbConn"
>

  <!-- add your bean definitions here! -->

</beans>
```

Lifecycle Annotations

JSR 250 LifeCycle Annotations

- **Applied to Methods**
  - **@PostConstruct**
  - **@PreDestroy**

```
public class InvoiceDAOImpl implements InvoiceDAO {
   private DatabaseConnection connection;

   @PostConstruct
   public void initializeDbConn() {
      connection = DatabaseConnection.getInstance();
   }

   public void persistInvoice(Invoice inv) {
      // Save the invoice in the database
   }

   @PreDestroy
   public void releaseDbConn() {
      connection.releaseConnection();
   }
}
```

Lifecycle Interfaces

- **There are a number of interfaces a bean may implement in order to receive callbacks**
  - **ApplicationContextAware**
    Allows your bean to get a reference to the ApplicationContext
  - **InitializingBean**
    You write the init function
  - **DisposableBean**
    You write the destroy function
  - **BeanNameAware**
    Allows a bean to get its name
  - **BeanPostProcessor**
    Special interface allowing interaction with newly created beans before and/or after their initialization method is called

Lifecycle Interfaces

- **ApplicationContextAware**
  Allows your bean to get a reference to the ApplicationContext
- **InitializingBean**
  You write the init function

```
public class IntlShippingChargeCalcImpl implements
      ShippingChargeCalculator, ApplicationContextAware,
      InitializingBean
{
  private String shipper;
  private ApplicationContext ctx;

  public double shippingCharge(Order order) {
    return order.getAmount() * .10;
  }

  // Needed for ApplicationContextAware interface
  public void setApplicationContext(
       ApplicationContext context) throws BeansException
  {
       this.ctx = context;
  }

  // Needed for InitializingBean interface
  public void afterPropertiesSet() throws BeansException {
      // Do any initializations here
  }
}
```

Lifecycle Interfaces

- **Generally you want to avoid using the Spring lifecycle interfaces**
  - **Their use makes your code Spring-specific**
- **Note that @PostConstruct and @PreDestroy don't tie your code to Spring**

LifeCycle Callbacks with Configuration

- **Instead of expressing Life Cycle callbacks in XML, it can also be done in Java Configurations**

```
package nvz.configurations;

import nvz.services.*;
import org.springframework.context.annotation.*;

@Configuration
public class BeanContext {
  @Bean(initMethod="doInit", destroyMethod="doCleanup")
  public ShippingChargeCalculator californiaShippingCalculator() {
    ShippingChargeCalculator shipCalc =
                  new ShippingChargeCalculatorSimpleImpl();
    shipCalc.setShipper("UPS");
    return shipCalc;
  }
}
```

Summary of Bean Characteristics

- **A Bean is an instance of a Java class**
- **Changeable characteristics**
  - **Name (id)**
  - **Properties** (Simple or References)
  - **Dependencies** (may depend on other beans or be a dependency of other beans; can use **depends-on** attribute)
  - **Autowiring mode** (byName, byType)
  - **Scope** (singleton, prototype, session,etc.)
  - **Initialization Style** (eager, lazy)
  - **Initialization method**
  - **Destruction method**