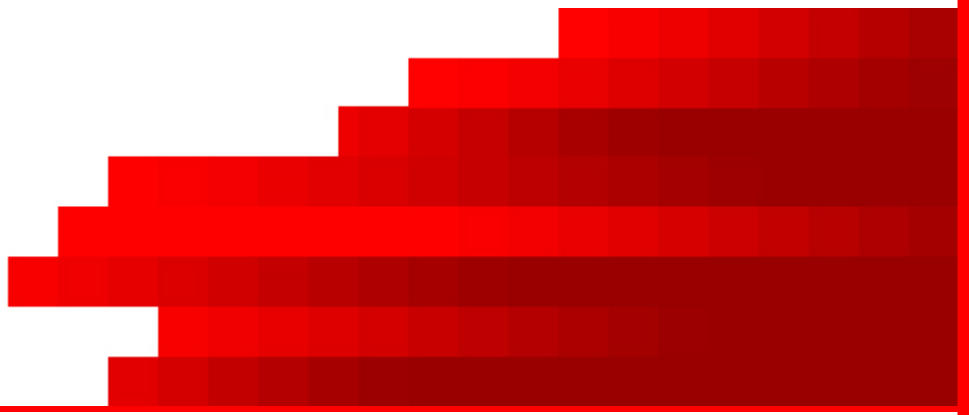
Spring framework



HSBC Technology and Services

HSBC
The world's local bank

Agenda

- Introduce Spring framework
- Dependency Injection (DI) Concept
- AoP Concept
- Spring IoC Container
- Spring Configurations

Spring Framework

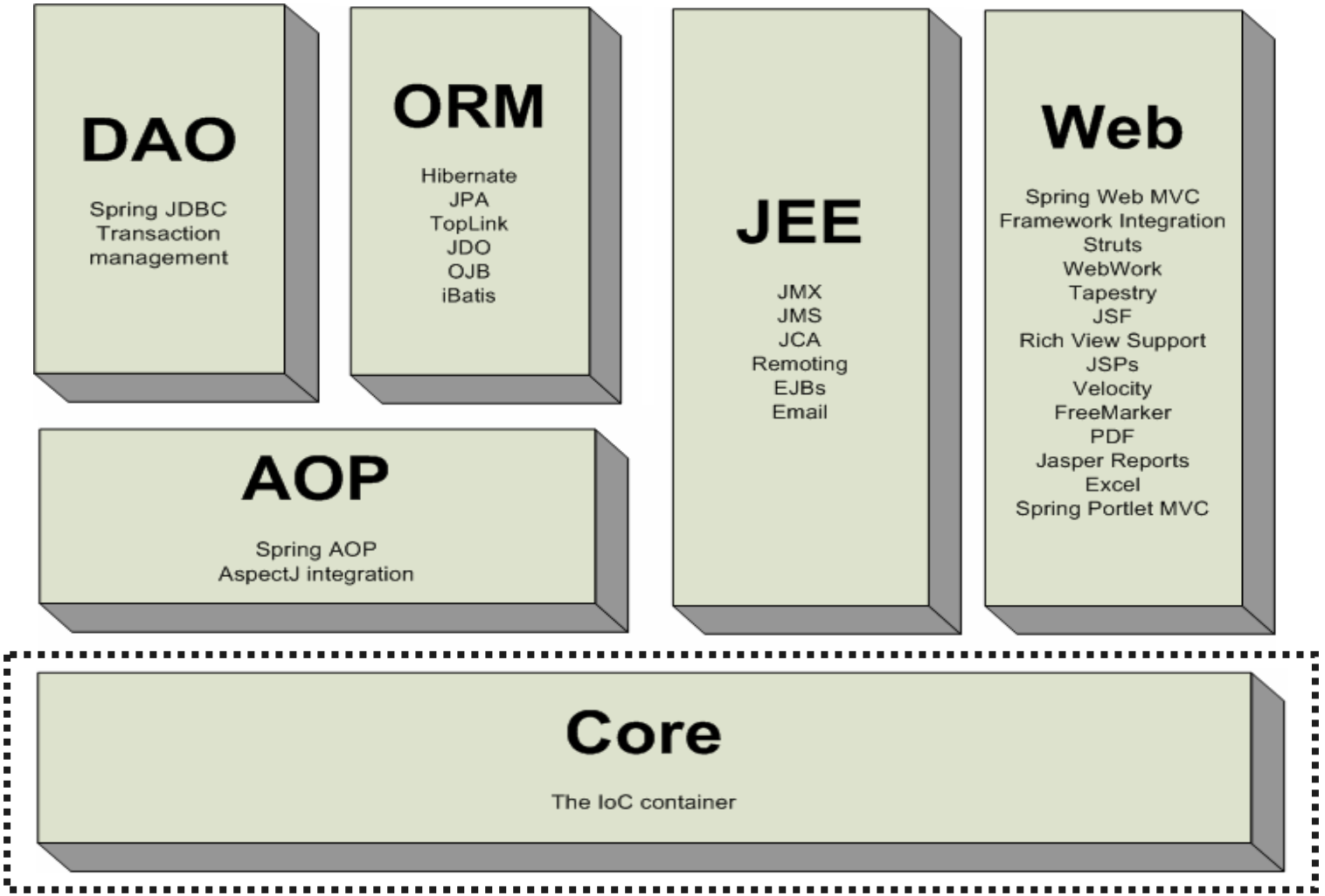
- Spring is an open source framework created to address the complexity of enterprise application development.
- Address end-to-end requirements rather than one tier.
- Enable Enterprise Java development while stick to old Java idioms, like interface-based programming and JavaBeans
- Introduced by Rod Johnson in “J2EE Design & Development”

Spring Framework

- Features of Spring

1. Lightweight: Spring is lightweight in terms of both size and overhead.
2. Spring is nonintrusive: objects in a Spring-enabled application often have no dependencies on Spring-specific classes.
3. Inversion of control framework that allows bean dependencies to be automatically resolved upon object instantiation
4. Support for Aspect-oriented Programming.
5. Spring is a container in the sense that it contains and manages the lifecycle and configuration of application objects.
6. Fully portable across deployment environments

Spring Framework



Dependency injection

Introduction to Dependency Injection

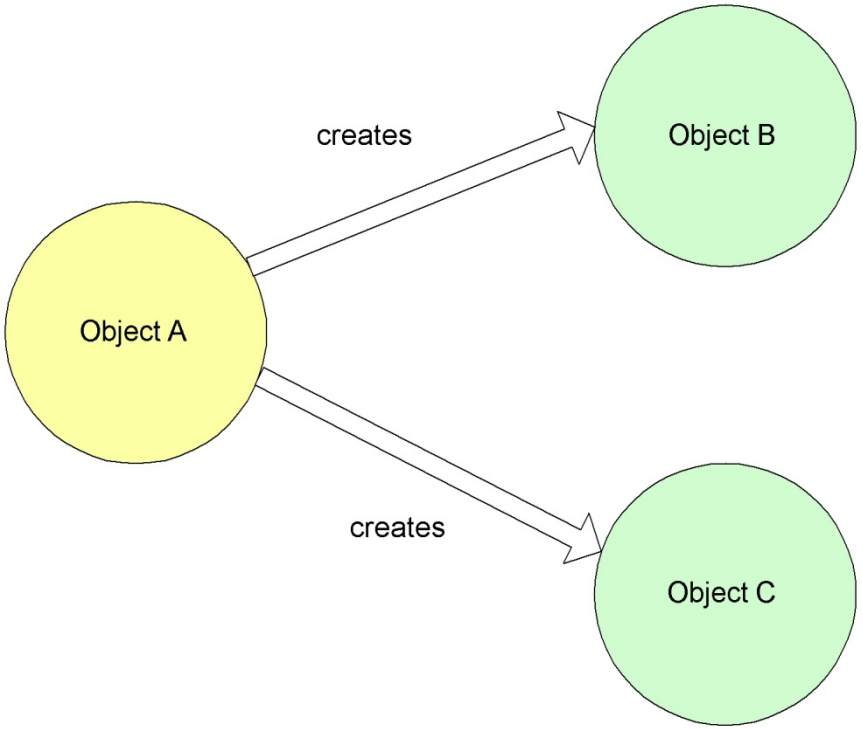
- **What is it?**

  A technique in which objects are passively given their dependencies instead of creating or looking for dependent objects for themselves

Dependency injection

- Originally, commonly referred to by another name:inversion of control.
- Based on Hollywood Principle: "Don't call me, I'll call you.“
- It’s a form of push configuration.
- You don't need to do anything in particular to make an application class eligible for Dependency Injection.

Non-IoC / Dependency Injection

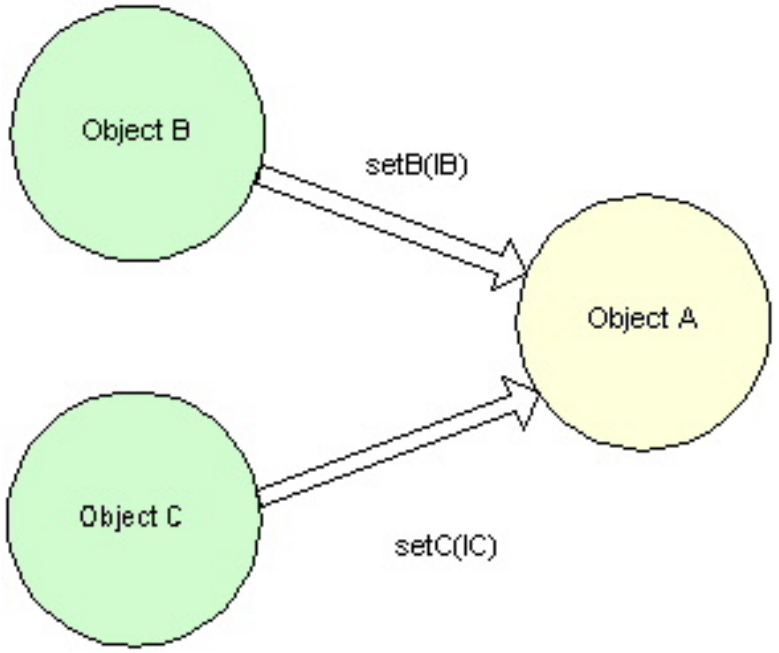


Non-IoC / Dependency Injection

```
public class AccountDaoImpl implements AccountDao {

    AccountRepository  accountRepository = null;

   public Account getAccountDetails(String accountNo)  {
      accountRepository = AccountRepository.getInstance();
    return accountRepository . getAccountDetails(accountNo);

}
```

HSBC
The world's local bank

With Dependency injection



Dependency injection

```
public class AccountDaoImpl implements AccountDao {

   AccountRepository  accountRepository = null;

  public Account getAccountDetails(String accountNo)  {
     return getAccountRepository(). getAccountDetails(accountNo);

  }

 public AccountRepository getAccountRepository() {
    return accountRepository;
 }

public void setAccountRepository(AccountRepository accountRepository) {
     this.accountRepository = accountRepository;
}
```

Dependency injection

- Types of Dependency Injection

1. **Setter Injection - The injection of dependencies via JavaBean setter methods.**

```
public void setName (String name){
      this.name = name
}
```

2. **Constructor Injection: The injection of dependencies via constructor arguments**

```
public Test (String name){
     this.name = name
}
```

HSBC
The world's local bank

Dependency injection

- Benefits
- **Removes the responsibility of finding or creating dependent objects and moves it into configuration**
- **Reduces coupling between implementation objects and encourages interface based design**
- **Allows an application to be reconfigured outside of code**
- **Can encourage writing testable components**
- **Improves testability**
  - **Dependencies can be easily stubbed out for unit testing**

HSBC
The world's local bank

Aspect Oriented Programming - AoP

Aspect Oriented Programming (AoP)

- Applications must be concerned with things like:
  - Logging
  - Transaction management
  - Security
- These concerns often find their way into application components whose core responsibility is something else.
- Such concerns are often referred to as “cross cutting” concerns

HSBC
The world’s local bank

Aspect Oriented Programming(AoP)

- Cross Cutting concerns introduces following problem:
  - Code that implements system wide concerns is duplicated across multiple components
  - Your components are littered with code that isn't aligned with their core functionality.

Aspect Oriented Programming (AoP)

- Aspect-oriented programming is a programming technique that promotes separation of concerns within a software system.
- AOP makes it possible to modularize these services and then apply them declaratively to the components that they should affect.
- AoP ensure that POJOs remain plain.

Aspect Oriented Programming(AoP)

Some AOP concepts

- Aspect - a modularization of a concern that cuts across multiple classes.

  E.g. Logging

  Aspects are implemented using regular Java classes

- Join point: A joinpoint is a point in the execution of the application such as the execution of a method or the handling of an exception, where an aspect can be plugged in.

- PointCut: Pointcuts help narrow down the joinpoints advised by an aspect.

- Advice: action taken by an aspect at a particular join point.
  - **Before Advice**
  - **After Advice**
  - **Around Advice**
  - **Throws Advice**

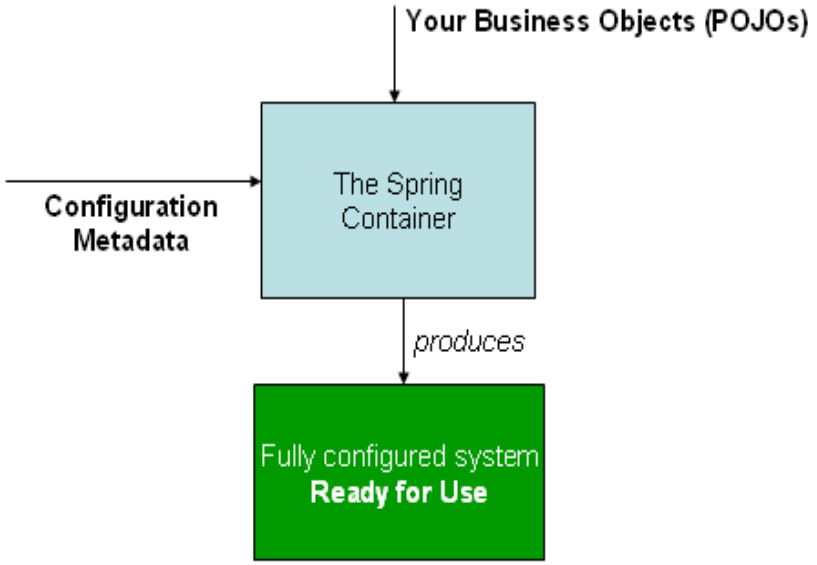
HSBC
The world's local bank

THE SPRING BEAN CONTAINER

Spring IoC Container

- The Spring IoC container is at the core of the Spring Framework.
- Uses IoC to manage components that make up an application
- Components are expressed as regular Java Beans
- Container manages the relationships between beans and is responsible for configuring them
- Manages the lifecycle of the beans

Spring IoC Container



- **The Spring IoC container consumes some form of *configuration metadata***
- *Configuration metadata:*
  - Spring beans, dependencies, and the services needed by beans are specified in here
  - nothing more than how to inform the Spring container as to how to "instantiate, configure, and assemble the objects in an application"

HSBC
The world's local bank

Spring IoC Container

- Types of Bean Containers
- Bean Factory
  - **Provides basic support for dependency injection**
  - **Configuration and lifecycle management**
  - **org.springframework.beans.factory.BeanFactory is actual representation of Spring IoC container**
  - ***org.springframework.beans.factory.xml.XmlBean-Factory* is most commanly used implementation**

Spring IoC Container

- Types of Bean Containers
- Application Context

  Builds on Bean Factory and adds services for

  - **Resolving messages from properties files for internationalization**
  - **Loading generic resources**
  - **Publishing events**

Spring IoC Container

- Among the many implementations of ApplicationContext are three that are commonly used:

1. **ClassPathXmlApplicationContext: Loads a context definition from an XML file located in the classpath, treating context definition files as classpath resources.**
2. **FileSystemXmlApplicationContext: Loads a context definition from an XML file in the file system.**
3. **XmlWebApplicationContext: Loads context definitions from an XML file contained within a web application.**

```
ApplicationContext context = new FileSystemXmlApplicationContext("c:/foo.xml");

ApplicationContext context = new ClassPathXmlApplicationContext("foo.xml");
```

Spring IoC Container

- Configuring an XMLWebApplicationContext

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
                /WEB-INF/applicationContext.xml
        </param-value>
</context-param>

<listener>
        <listener-class> org.springframework.web.context.ContextLoaderListener
        </listener-class>
</listener>
```

HSBC
The world's local bank

Spring Configurations

Spring Configurations

Spring Configurations – Java Bean

Example: *com.hsbc.beans.SimpleBean* model class

```
public class SimpleBean {
private String name = null;
private int staffId = 0;

public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public SimpleBean(){
}
public SimpleBean(String name, int staffId){
this.name = name;
this.staffId = staffId;
}
public SimpleBean(String name){
this.name = name;
}
public SimpleBean(int staffId){
this.staffId = staffId;
}
public int getStaffId() {
return staffId;
}
public void setStaffId(int staffId) {
this.staffId = staffId;
}
}
```

HSBC
The world's local bank

Spring Configurations: Setter Method Injection

Demo.xml declares the instance of *SimpleBean* in the spring container

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

  <bean id='glt001' class='com.hsbc.beans.SimpleBean'>



        <property name='name' value='xyz'/>
        <property name='staffId' value='10551328'/>

</bean>
</beans>
```

Root element

"SimpleBean" class info

Set "name" & "staffId" property

Spring Configurations: Setter Method Injection

- <beans> element
  - The root of the demo.xml file
  - The root element of any Spring configuration file
- <bean> element
  - Tells the Spring container about the class and how it should be configured
  - ***id*** attribute takes the unique id
  - ***class*** attribute specifies the bean's fully qualified class name

```
<bean id='glt001' class='com.hsbc.beans.SimpleBean'>
```

HSBC
The world's local bank

Spring Configurations: Setter Method Injection

- <property> element
  - Tells the Spring container to call setName(), setStaffId while setting the “name” and “staffId” property
- <value> element
  - Defines the value of “name” and “staffId” as “xyz” and “10551328”
- **The container instantiates the 'SimpleBean' based on the XML definition as:**

```
SimpleBean glt001 = new SimpleBean();
glt001. setName(“xyz");
glt001.setStaffId (“10551328");
```

HSBC
The world’s local bank

Spring Configurations: Constructor Injection

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

  <bean id='glt001' class='com.hsbc.beans.SimpleBean'>



        <constructor-arg value='xyz'/>
        <constructor-arg value='10551328'/>

</bean>

</beans>
```

Root element

"SimpleBean" class info

Set "name" and "staffId" property through constructor

HSBC
The world's local bank

Spring Configurations: Constructor Injection

- **The container instantiates the ‘SimpleBean' based on the XML definition as:**

```
SimpleBean glt001 = new SimpleBean(“xyz”, 10551328);
```

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("demo.xml");

SimpleBean bean = (SimpleBean) ctx.getBean("glt001");
```

Spring Configurations: Constructor Injection

- Different forms of Constructor Injection

- Constructor Argument Type Matching

```
<bean id='glt003' class='com.hsbc.beans.SimpleBean'>
    <constructor-arg type='int' value='35027901'/>
    <constructor-arg type='java.lang.String' value='lmn'/>
 </bean>
```

- Constructor Argument Index

```
<bean id='glt003' class='com.hsbc.beans.SimpleBean'>
    <constructor-arg index='0' value='qwer'/>
    <constructor-arg index='1' value='26005401'/>
</bean>
```

HSBC
The world's local bank

Spring Configurations

- Combining Constructor and Setter Injection

```
<bean id='glt005' class='com.hsbc.beans.SimpleBean'>
     <constructor-arg type='int' value='43197070'/>
     <property name='name' value='qaz'/>
 </bean>
```

Spring Configurations: Ref-Bean

<ref bean> element

- **Used to set properties that reference other beans**
- **Defined as sub-element of <property> element**

```
<?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">

  <beans>
      <bean id=“cars"   class=“com.spring.demo.Cars">
          <property name=“myCar“ >
              <ref bean=“nano”/>
          </property>
      </bean>
      <bean id=“nano"   class=“com.spring.demo.Nano" />
</beans>
```

HSBC
The world's local bank

Spring Configurations: Inner Bean

- **A <bean/> element inside the <property/> or <constructor-arg/> elements**
- **Inner beans do not have an id attribute.**

```
<?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">

  <beans>
        <bean id=“cars"   class=“com.spring.demo.Cars">
              <property name=“myCar“ >
                    <bean class=“com.spring.demo.Nano" />
              </property>
        </bean>
</beans>
```

HSBC
The world’s local bank

Spring Configurations: Inner Bean

- **Inner beans in constructor injection**

```
<?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">

  <beans>
        <bean id=“cars"   class=“com.spring.demo.Cars">
              <constructor-arg>
                    <bean class=“com.spring.demo.Nano" />
              </ constructor-arg>
        </bean>
</beans>
```

HSBC
The world’s local bank

Spring Configurations: Compound Properties

```
<bean id="country" class="example.India">
                    <property name="person.age" value="23" />
</bean>
```

- **“country” bean must have a “person” nested bean, which in turn has an “age” property**
- **Equivalent to *countryBean.getPerson().setAge(23)***

HSBC
The world’s local bank

Spring Configurations: Aliasing beans

- **Bean Ids are unique within the container.**
- **A bean will have almost only one Id and one or more aliases.**
- **Alias can be specified by name attribute of <bean> tag**

```
<beans>
    <bean id="cars"   class="com.spring.demo.Cars" name="cars1, cars2" />
</beans>
```

- **Alias can be specified by <alias> tag**

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

HSBC
The world's local bank

Spring Configurations: Bean Scope

- **“scope” attribute of <bean> tag define the bean scopes.**
- **Possible values of “scope” attribute are:**
  - **singleton**
  - **prototype**
  - **request**
  - **session**
  - **global session**
- **By default, all Spring beans are singletons.**
- **Spring's singleton beans only guarantee a single instance of the bean definition per the application context.**

```
<bean id=“cars"   class=“com.spring.demo.Cars“ name=“cars1; cars2”  scope = “prototype”/>
```

HSBC
The world’s local bank

Spring Configurations: Lazily-instantiated beans

- **Applicable only for singleton beans.**
- **Beans are “lazily” loaded into bean factories.**
  - beans will not be instantiated until they are needed.
- **By default ApplicationContext eagerly pre-instantiate all singleton beans at start-up.**
- **lazily-initialized bean Indicates whether or not a bean instance should be created at start-up or when it is first requested.**

```
<bean id=“cars"   class=“com.spring.demo.Cars“ lazy-init="true"/>
```

- **lazy-initialisation at the container level can be controlled by using the 'default-lazy-init' attribute on the <beans/>**

```
<beans default-lazy-init="true">
```

HSBC
The world’s local bank

Spring Configurations: Wiring collections

- **Spring allows for injecting following kinds of collections:**

Collection element	Corresponding Java elements	Useful for…
<list>	Array or java.util.List	Wiring a list of values in array or Java.util.List, allowing duplicates.
<set>	Array or java.util.Set	Wiring a set of values in array or Java.util.Set, ensuring no duplicates.
<map>	java.util.Map	Wiring a collection of name-value pairs where name and value can be of any type
<props>	java.util.Properties	Wiring a collection of name-value pairs where the name and value are both Strings

Spring Configurations: Wiring collections

- **<list>**

```
<property name ='cars'>
   <list>
      <value>SWIFT</value>
      <value>BMW</value>
       <ref bean =‘nano'/>
      <list>
        <value>ALTO</value>
        <value>ZEN</value>
      </list>
   </list>
</property>
```

- **<list> can be used to inject the java.util.List, java,util.Set or array elements**

```
private List cars;   Or
private Set cars;    Or
private Car[] cars;
```

HSBC
The world's local bank

Spring Configurations: Wiring collections

- **<set>**

```
<property name ='cars'>
   <set>
      <value>SWIFT</value>
      <value>BMW</value>
       <ref bean ='nano'/>
      <list>
        <value>ALTO</value>
        <value>ZEN</value>
      </list>
   </set>
</property>
```

- **<set> can be used to inject the java.util.List, java.util.Set or array elements**

```
private List cars;   Or
private Set cars;    Or
private Car[] cars;
```

HSBC
The world's local bank

Spring Configurations: Wiring collections

- **<map>**

```
<property name ='staff'>
    <map>
        <entry key ='xyz' value-ref='glt001' />
        <entry key-ref='glt003' value='11'/>
    </map>
</property>
```

- **An <entry> in a <map> is made up of a key and a value, either of which can be a primitive value or a reference to another bean.**

Attribute	Purpose
key	Specifies the key of the map entry as a String
key-ref	Specifies the key of the map entry as a reference to a bean in the Spring context
value	Specifies the value of the map entry as a String
value-ref	Specifies the value of the map entry as a reference to a bean in the Spring context

HSBC
The world's local bank

Spring Configurations: Wiring collections

- **<props> used to define a collection value of type java.util.Properties.**

```
<property name='accounts'>
   <props>
      <prop key="HSBC">123456</prop>
      <prop key="HDFC">456789</prop>
   </props>
</property>
```

- `<prop>` **- used to define a member value of a** `<props>` **collection.**

- `key` **– defines the key of each "Properties" member .**

- **Contents of** `<prop>` **defines the value of "Properties" member.**

HSBC
The world's local bank

Spring Configurations: Creating beans from static factory methods

- `factory-method` **attribute of** `<bean>` **lets you specify a static method to be invoked instead of the constructor to create an instance of a class.**

```
<bean id='accountRepository'
  class='com.hsbc.hsbcnet.beans.AccountRepository' factory-
  method='getInstance' />
```

- `factory-method` **- perfectly suitable for any occasion where you need to wire an object produced by a static method.**

HSBC
The world's local bank

Spring Configurations: Creating beans from instance factory methods

- **a non-static method of an existing bean from the container is invoked to create a new bean.**
- **To use instance factory method:**

- `class` **attribute must be left empty**
- `factory-bean` **attribute must specify the name of a bean in the container**
- `factory-method` **must specify name of the factory method**

```
<bean id='accountRepository'
  class='com.hsbc.hsbcnet.beans.AccountRepository'/>

<bean id='accounts' factory-bean='accountRepository'
                    factory-method='getInstance' />
```

HSBC
The world's local bank

Spring Configurations: Parent and child beans

- **Several individual `<bean>` declarations can make Spring configuration unwieldy and brittle.**

```
<bean id='glt001' class='com.hsbc.glt.Staff'>
        <property name='staffId' value='glt001'/>
        <property name='name' value='Abc'/>
        <property name='dept' value='GTB'/>
        <property name='location' value='GLT 2.0'/>
    </bean>

    <bean id='glt002' class='com.hsbc.glt.Staff'>
        <property name='staffId' value='glt002'/>
        <property name='name' value='Xyz'/>
        <property name='dept' value='GTB'/>
        <property name='location' value='GLT 2.0'/>
    </bean>

    <bean id='glt003' class='com.hsbc.glt.Staff'>
        <property name='staffId' value='glt003'/>
        <property name='name' value='Pqr'/>
        <property name='dept' value='GTB'/>
        <property name='location' value='GLT 2.0'/>
    </bean>
```

HSBC
The world's local bank

Spring Configurations: Parent and child beans

- **Spring provides the facility to create `<bean>`s that extend and inherit from other `<bean>` definitions.**

```
<bean id='gtb' class='com.hsbc.glt.Staff'
  abstract='true'>
    <property name='dept' value='GTB'/>
    <property name='location' value='GLT 2.0'/>
</bean>

<bean id='glt001' parent='gtb'>
    <property name='staffId' value='glt001'/>
    <property name='name' value='Abc'/>
</bean>
<bean id='glt002' parent='gtb' >
    <property name='staffId' value='glt002'/>
    <property name='name' value='Xyz'/>
</bean>
<bean id='glt003' parent='gtb' >
    <property name='staffId' value='glt003'/>
    <property name='name' value='Pqr'/>
</bean>
```

HSBC
The world's local bank

Spring Configurations: Parent and child beans

- `<bean>` **element provides two special attributes for sub-beaning:**

- `parent`: **Indicates the id of a <bean> that will be the parent of the <bean> with the parent attribute.**

- `abstract`: **If set to true, indicates that the bean never be instantiated by Spring.**

- **Parent bean doesn't have to be abstract.**
- **Child bean can override the inherited properties**

```
<bean id='glt004' parent='gtb' >
    <property name='staffId' value='glt003'/>
    <property name='name' value='Qaz'/>
   <property name='location' value='GLT 1.0'/>
</bean>
```

HSBC
The world's local bank

Spring Configurations: Parent and child beans

- **Abstracting common properties**

```
<bean id='gtb' abstract='true'>
    <property name='dept' value='GTB'/>
    <property name='location' value='GLT 2.0'/>
</bean>

<bean id='glt001' class='com.hsbc.glt.Staff' parent='gtb'>
    <property name='staffId' value='glt001'/>
    <property name='name' value='Abc'/>
</bean>

<bean id='hsncNet' class='com.hsbc.glt.Project'
  parent='gtb'>
    <property name='projName' value='HSBCnet'/>
</bean>
```

HSBC
The world's local bank

Spring Configurations: Multiple Configuration Files

- The Spring Configuration XML can be split into multiple files
- The reference of one XML configuration can be given to another by using
  - Import
  - Application Context (programmatically)
  - Listeners (declaratively)

Spring Configurations: Multiple Configuration Files: Import

- **An XML can use** <import> **tag to import bean definitions defined in external XML**

```
<beans>

   <import resource='springconfig.xml' />

   <import resource="resources/dao.xml"/>

</beans>
```

- **The end result of these import statement is same as those defined inline in one XML**

Spring Configurations: Multiple Configuration Files: Application Context

- **Application Context**

```
ApplicationContext context = new ClassPathXmlApplicationContext(
                        new String[] {"services.xml", "dao.xml"});
```

Spring Configurations: Multiple Configuration Files: Declaratively

- **Declarative declaration of application contexts through a** ContextLoaderListener **also supports multiple XMLs to be loaded**

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/webContext.xml /WEB-INF/sevices.xml</param-value>
</context-param>
```

File 1 *File 2*

```
<listener>
    <listener-class>
            org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

HSBC
The world's local bank

Spring Configurations: Init-Method

InitializingBean interface

- **Allows a bean to perform initialisation work after all necessary properties on the bean are set by the container**

```
public class AnotherExampleBean implements InitializingBean  {
    public void afterPropertiesSet( )  {
      // do some initialization work
   }
}
```

- **Is discouraged since it unnecessarily couples the code to Spring**

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

HSBC
The world's local bank

Spring Configurations: Init-Method

Generic initialisation method

- `init-method` **attribute of** `<bean>` **provide the init method definition**

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

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

- **Does exactly the same job as *InitializingBean* interface.**
- `default-init-method` **attribute of** `<beans>` **provide the default init method for all the bean definitions.**

```
<beans default-init-method ='init'>
</beans>
```

HSBC
The world's local bank

Spring Configurations: Destroying beans

DisposableBean interface

- **Allows a bean to perform all necessary cleanup before a bean is removed from the container.**

```
public class AnotherExampleBean implements DisposableBean {
    public void destroy ( ) {
      // do some cleanup work
   }
}
```

- **Is discouraged since it unnecessarily couples the code to Spring**

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

HSBC
The world's local bank

Spring Configurations: Destroying beans

Destroy method

- `destroy-method` **attribute of** `<bean>` **provide the init method definition**

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

```
public class ExampleBean {
    public void destroy( )  {
        // do some initialization work
    }
}
```

- **Does exactly the same job as *DisposableBean* interface.**
- `default-destroy-method` **attribute of** `<beans>` **provide the default destroy method for all the bean definitions.**

```
<beans default-destroy-method ='destroy'>
</beans>
```

HSBC
The world's local bank

Spring Configurations: Autowiring

- `autowire` **attribute of** `<bean>` **allows spring container to autowire relationships between collaborating beans.**
- **By default, beans will not be autowired unless you set the** `autowire` **attribute.**
- **Spring provides four flavours of autowiring:**
  - **byName**
  - **byType**
  - **constructor**
  - **autodetect**

HSBC
The world's local bank

Spring Configurations: Autowiring

- **byName –**
- **Autowiring by property name.**
- **Limitation: It assumes that the name of a bean is same as name of the property of another bean that is going to be injected.**

Spring Configurations: Autowiring

- **byType –**
- **Allows a property to be autowired if there is exactly one bean of the property type in the container.**
- **If more than one bean matches, an exception will be thrown.**

Spring Configurations: Autowiring

- **constructor –**
- **This is analogous to *byType*, but applies to constructor arguments.**
- **If more than one bean matches, an exception will be thrown.**
- **Not suitable for a class having multiple constructor and any of which can be satisfied by autowiring.**

Spring Configurations: Autowiring

- **autodetect –**
- **Chooses constructor or byType through introspection of the bean class**
- **If a default constructor is found, the byType mode will be applied.**

Spring Configurations: Autowiring

- **Mixing auto with explicit wiring**
- **Explicit dependencies in property and constructor-arg settings always override autowiring.**
- **you cannot mix `<constructorarg>` elements with constructor autowiring.**
- **`<null/>` can be used to force an autowired property to be null.**

```
<property name='name'><null/></property>
```

Spring Configurations: Postprocessing beans

- Postprocessing allows us to cut into a bean's lifecycle and review or alter its configuration.
- Postprocessing trigged while instantiation and configuration of a bean.
- Post Processor must Implement `org.springframework.beans.factory.config.BeanPostProcessor` interface.

Spring Configurations: Postprocessing beans

```
public interface BeanPostProcessor {

Object postProcessBeforeInitialization(Object bean, String name)throws
  BeansException;

 Object postProcessAfterInitialization(Object bean, String name)throws
  BeansException;

}
```

- `postProcessBeforeInitialization` - **called immediately prior to bean initialization**
- `postProcessAfterInitialization` – **called immediately after initialization.**

HSBC
The world's local bank

Spring Configurations: Postprocessing the bean factory

- **A BeanFactoryPostProcessor performs postprocessing on the entire Spring container.**

```
public interface BeanFactoryPostProcessor {

    void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)

         throws BeansException;

}
```

- `postProcessBeanFactory` **– by the Spring container after all bean definitions have been loaded but before any beans are instantiated.**
- **Only available with application context containers.**

Spring Configuration: PropertyPlaceholderConfigurer

- A bean factory post-processor
- It would be used to externalise property values from a BeanFactory definition into another separate file in the standard Java Properties format
- This is useful to allow deploying an application to customize environment-specific properties.

Spring Configuration: PropertyPlaceholderConfigurer

- **The location property *PropertyPlaceholderConfigurer* of tells Spring where to find the property file.**
- **The location property allows you to work with a single property file.**

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

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

HSBC
The world's local bank

Spring Configuration: PropertyPlaceholderConfigurer

- ***locations* property of *PropertyPlaceholderConfigurer* allows to set a List of property files.**

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

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

HSBC
The world's local bank

Spring Configuration: Resolving text messages

- **Spring's *ApplicationContext* supports parameterized messages by making them available to the container through the *MessageSource* interface**
- ***ResourceBundleMessageSource* implementation simply uses Java's own java.util.ResourceBundle to resolve messages.**

```
<bean id="messageSource"  class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename">
            <value>trainingtext</value>
    </property>
 </bean>
```

- **bean be named must be messageSource**

HSBC
The world's local bank

Spring Configuration: Resolving text messages

- You'll never need to inject the *messageSource* bean into your application
- use ApplicationContext' own getMessage() methods to access the message from *messageSource.*

Spring Configuration: Best Practice

- Prefer setter injection over constructor injection
- Avoid using autowiring
- Prefer Type over Index for constructor argument matching.
- Reuse bean definitions, if possible.
- Do not abuse dependency injection

Resources

- Resources
  - Spring in Action 2nd Edition
  - Spring Reference Guide

    http://www.springframework.org/documentation
  - E-Campus Course: Prism4Java 6.0 Prerequisite: Core Spring Framework
  - Professional Java Development with the Spring Framework by Rod Johnson

Q & A

Thank You!

Any Questions?

HSBC
The world's local bank