Spring Framework

Vivek S. Kulkarni
Ramkrishna IT Consulting Pvt. Ltd.
9860632616

Core Spring : Objectives

- Understand the architecture of Spring Framework
- Understand Dependency Injection
- AOP
- Develop sample applications using
  - Spring AOP
  - Spring JDBC Support
  - Spring Hibernate integration
  - Transactional Support in Spring
  - Spring Remoting
  - Spring MVC

Core Spring : Day 1

- Installing and Using STS
- Writing a simple Spring app
- Understanding ApplicationContext
- Learn bean lifecycle
- Learning Dependency Injection
  - Constructor injection
  - Setter Injection
- Annotation based DI

Course Agenda: Day 2

- AOP : Advice, JoinPoints, PointCuts and Aspects
- Adding behavior to an application using aspects
- Introducing data access with Spring
- Simplifying JDBC-based data access
- Driving database transactions in a Spring environment

Course Agenda: Day 3

- Introducing object-to-relational mapping (ORM)
- Getting started with Hibernate in a Spring environment
- Web application architecture
- Getting started with Spring MVC
- RESTful WebServices Support in Spring

Overview of the Spring Framework

Introducing Spring in the context of enterprise application architecture

What is Spring ?

A light-weight dependency injection and aspect-oriented container and framework

What is Spring ?

Light-Weight : Processing overhead required by spring is negligible

Non-intrusive : Spring enabled application may not have any dependency on spring, They can be run inside or outside of the spring framework without any change in source code.

Dependency Injection : Promotes *program-to-interface* design principle, thus promotes loose coupling. This technique is reverse of JNDI –i.e. you do not look-up for dependant objects in your code, container figures out required objects and creates them and INJECTS them.

What is Spring ?

Aspect-oriented : Application programmers do not write logic for cross-cutting concerns such as logging, transaction, security. The Spring framework provides the support for AOP.

Container : Spring is a container in the sense that it contains and manages the lifecycle and configuration of application objects. The association of different application objects, their dependencies, their initialization etc. is configured and container does the rest.

Framework : A large number of pre-defined classes and interfaces provide a host of functionality required like transaction management, MVC, integration with persistence frameworks and so on. You write only business logic.

Goal of the Spring Framework

- Provide comprehensive infrastructural support for developing enterprise Java™ applications
  - Spring deals with the plumbing
  - So you can focus on solving the domain problem

Summary :

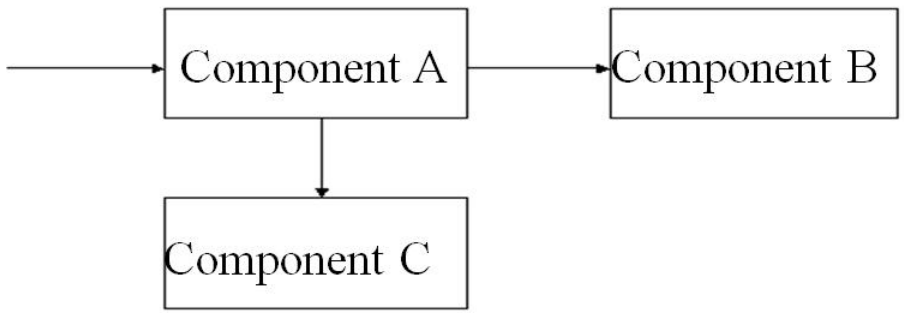
•Lightweight and minimally invasive development with plain old Java objects (POJOs).
•Loose-coupling through dependency injection and interface-orientation.
•Declarative programming through aspects and common conventions.
•Boilerplate reduction through aspects and templates

Spring's Support (1)

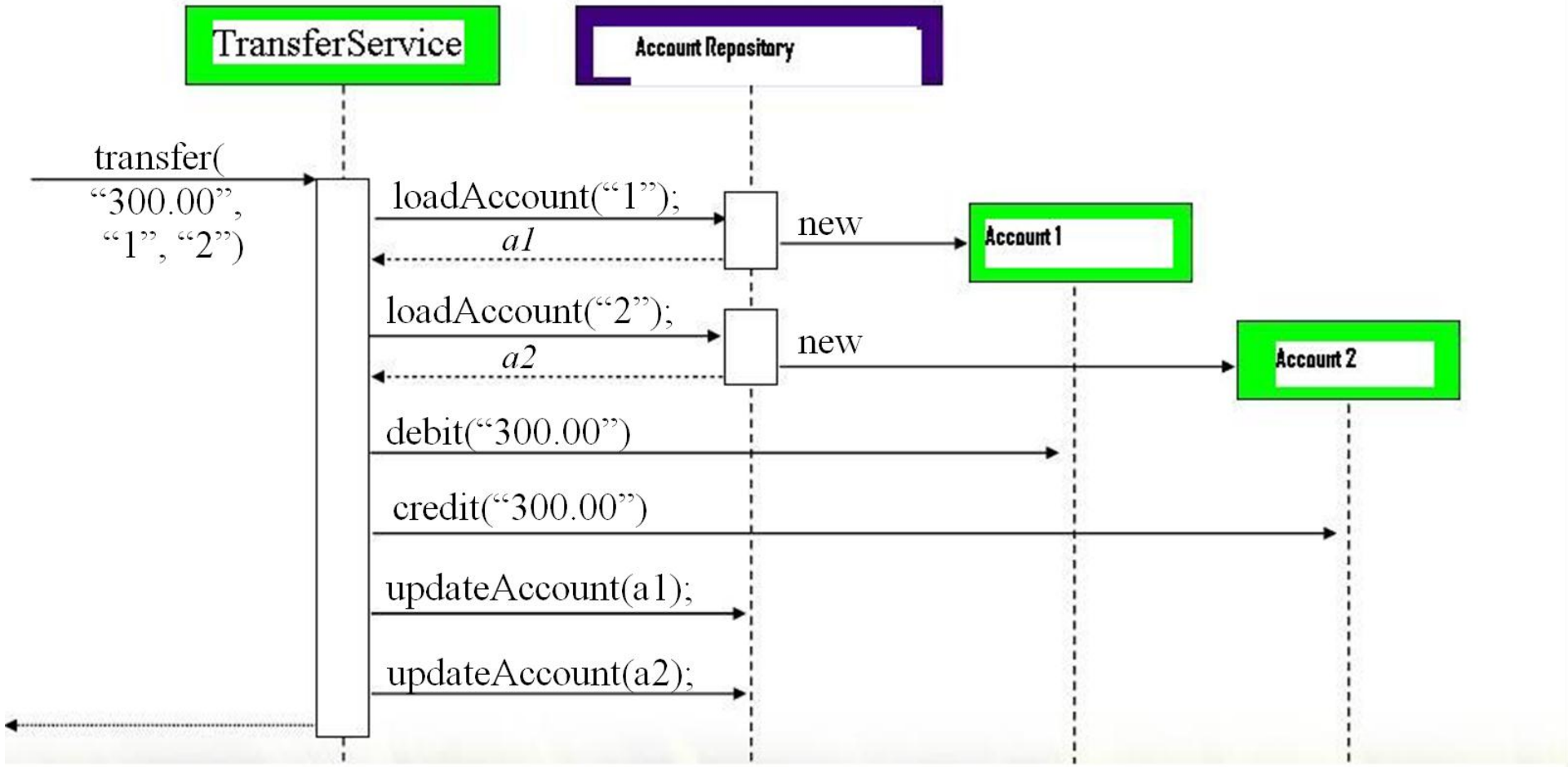
- Core support
  - Application Configuration
  - Enterprise Integration
  - Data Access

Application Configuration

- A typical application system consists of several parts working together to carry out a use case



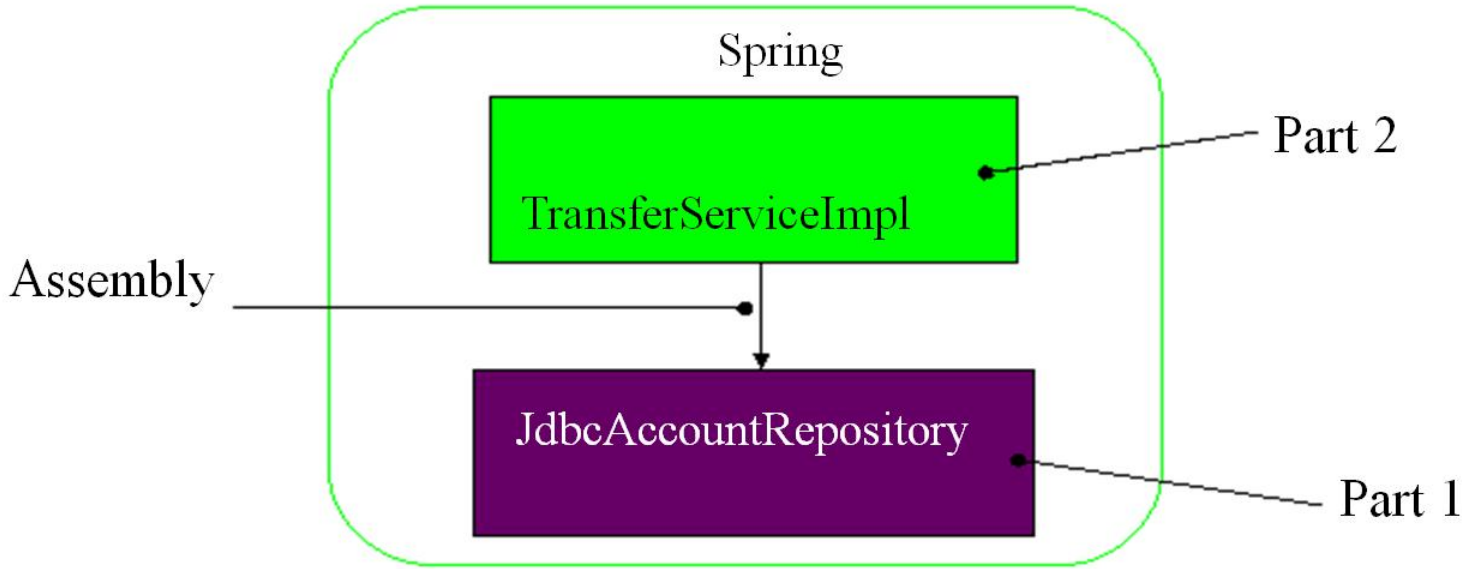
Use Case : Bank Money Transfer



Spring's Configuration Support

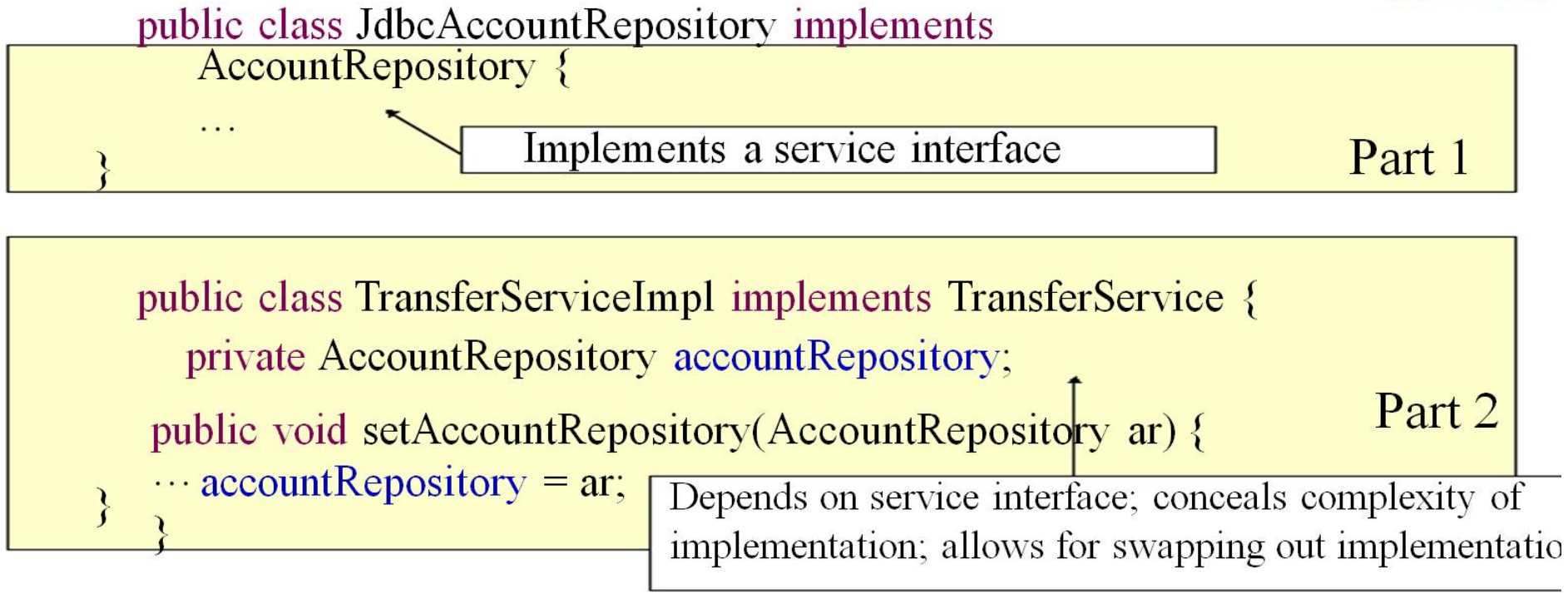
- Spring provides support for assembling such an application system from its parts
  - Parts do not worry about finding each other
  - Any part can easily be swapped out

Money Transfer System
Assembly




```
(1) new JdbcAccountRepository(…);
(2) new TransferServiceImpl();
(3) service.setAccountRepository(repository);
```

Parts are Just Plain Java Objects



Enterprise Integration

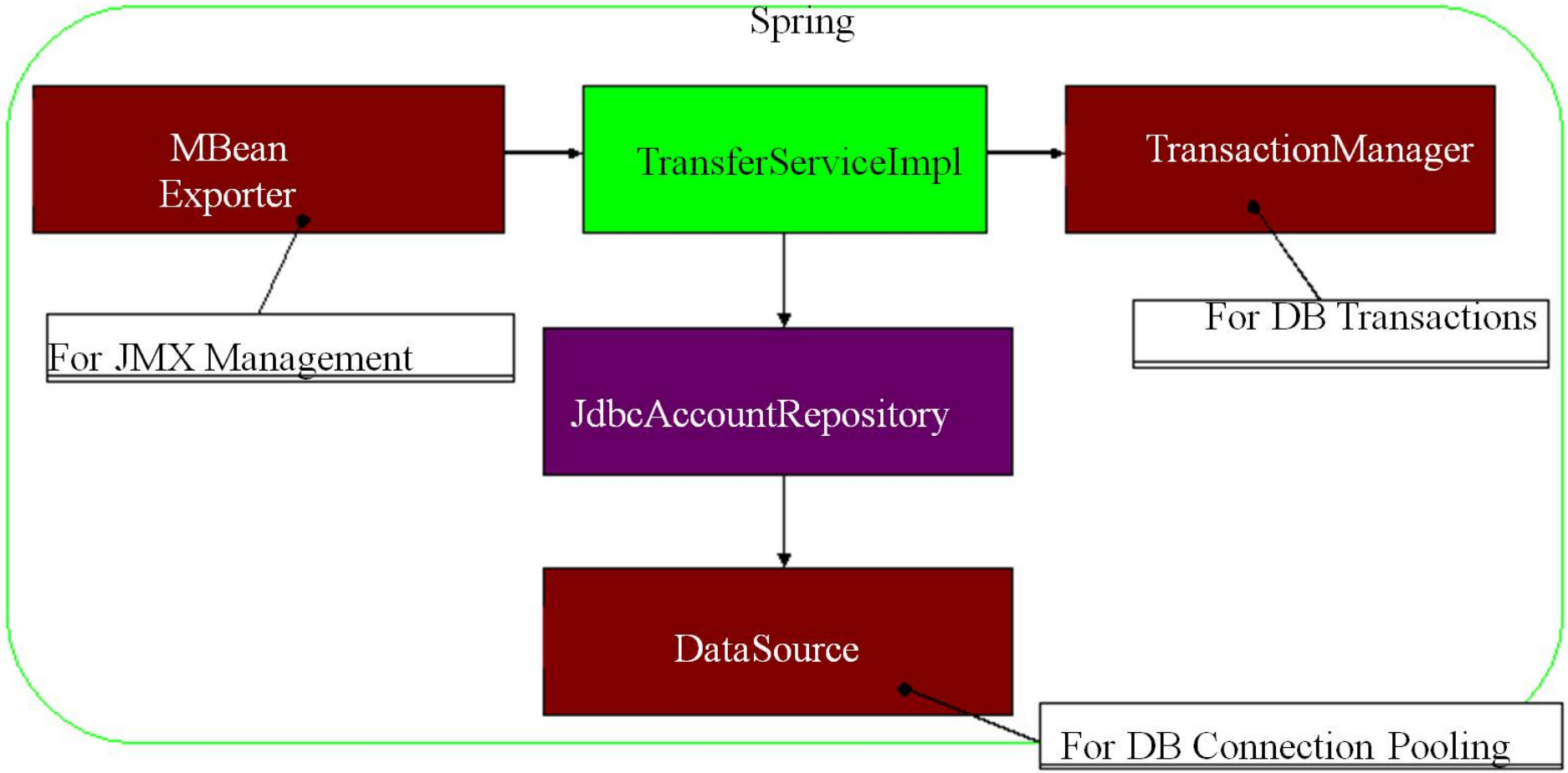
- Enterprise applications do not work in isolation
- They require enterprise services and resources

Database Connection Pool
Database Transactions
Security
Messaging
Remote Access
Caching

Spring Enterprise Integration

- Spring helps you integrate powerful enterprise services into your application
  - While keeping your application code simple and testable
- Plugs into all Java EE™ application servers
  - While capable of standalone usage

Enterprise Money Transfer with Spring



Simple Application Code


```
public class TransferServiceImpl implements TransferService {
    @Transactional
        public TransferConfirmation transfer(MonetaryAmount amount, in srcAccountId, int
        targetAccountId ){

        Account src = accountRepository.loadAccount(srcAccountId);
        Account target = accountRepository.loadAccount(targetAccountId);
        src.debit(amount);
        target.credit(amount);
        accountRepository.updateAccount(src);
        accountRepository.updateAccount(target);
        return new TransferConfirmation(...);
    }
}
```

Tells Spring to run this method
in a database transaction

Data Access

Spring makes data access easier to
- Manage resources for you
- Provide API helpers
- Support all major data access technologies

It also Supports Integration of
- Hibernate
- iBatis
- JPA

Spring JDBC in a Nutshell

Spring's JDBC helper class

```
int count = jdbcTemplate.queryForInt(
                    "SELECT COUNT(*) FROM CUSTOMER");
```

- Acquisition of the connection
- Participation in the transaction
- Execution of the statement
- Processing of the result set
- Handling of any exception
- Release of the connection

All handled by Spring

Spring Framework History

- http://www.springsource.org/download
- Spring 3.0 (released 12/09)
  - Requires Java 1.5+ and JUnit 4.7+
  - REST support, JavaConfig, SpEL, more annotations
- Spring 2.5 (released 11/07)
  - Requires Java 1.4+ and supports JUnit 4
  - Annotation DI, @MVC controllers, XML namespaces
- Spring 2.0 (released 10/06)
  - Compatible with Java 1.3+
  - XML simplification, async JMS, JPA, AspectJ support

Installing and Using STS

Download and install STS 2.7.x from SpringSource.com:

- Eclipse Based
- Has Spring Plug-ins
- Auto-Completion for Spring Config files
- Comes with Tomcat plugged-in
- Comes with Junit-plugged-in

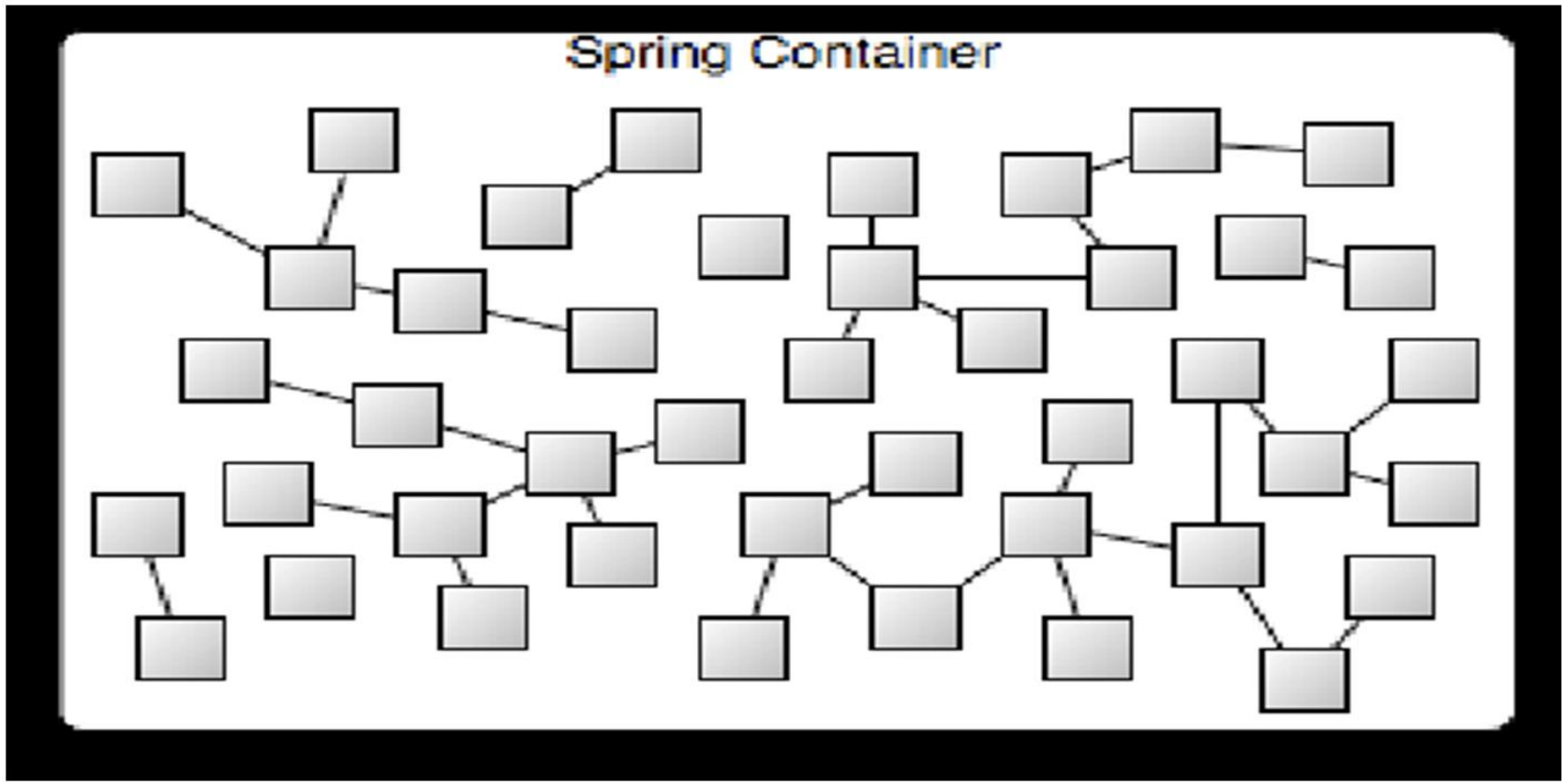
Installing and Using STS

Lab -1 : 30 minutes

Use STS for creating non-Spring App

Spring Container

- Cradle to Cremetory(new() to finalize())



Spring Container

- Cradle to Cremetory (new() to finalize())
- Old container(pre 3.x)
  org.springframework.beans.factory.BeanFactory interface
- New : org.springframework.context.ApplicationContext interface

Spring Container

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

Dependency Injection

Dependency Injection : Alternatives

- Creation using singletons
- Factory Pattern
- Reflection with Strategy Pattern
- Decorator
- Service Locator

Dependency Injection

Dependency Injection : Responsibility of co-ordination and collaboration is taken away from objects

- Decoupling : Program-to-Interface principle
- Dependency graph
- Clean code
- Enforcing immutability if required
- Enforcing initialization
- Runtime changes in dependency rules
- Objects focus on their task and not worried about availability of their dependent objects

Dependency Injection

Dependency Injection : Using Spring

Steps :

1. Identify the dependencies
2. Decide optional dependancies
3. Wire them together in XML file
4. Instantiate ApplicationContext(container)
5. Get Instance by calling getBean() method

Dependency Injection

Bank Service Example

Without Spring :

```
AccountRepository repos = new AccountRepositoryImpl();
Bank bank = new BankImpl(repos);
bank.withdraw(1234, 500);
```

```
ApplicationContext ctx = …
Bank bank = ctx.getBean("bankService");
bank.withdraw(1234, 500);
```

No need to instantiate AccountRepository, No knowledge of Bank Implementation class

Bank Service Example

Spring Bean Configuration File :

```
<bean id="bankService" class="bank.server.internal.BankImpl">
  <constructor-arg ref="accountRepo"/>

</bean>
<bean id="accountRepo" class="bank.server.internal.AccountRepositoryImpl">

</bean>
```

With Spring

```
ApplicationContext ctx = …
Bank bank = ctx.getBean("bankService");
bank.withdraw(1234, 500);
```

"bankService" must match with bean id

Bank Service Example

Spring Bean Configuration File :

```
<bean id="bankService" class="bank.server.internal.BankImpl">
  <constructor-arg ref="accountRepo"/>

</bean>
<bean id="accountRepo" class="bank.server.internal.AccountRepositoryImpl">

</bean>
```

Equivalent to

```
AccountRepository accountRepo = new bank.server.internal.AccountRepositoryImpl();
Bank Bank = new BankImpl(accountRepo);
```

Application Context

bankService	Instance of BankImpl
accountRepo	Instance of AccountRepositoryImpl

Constructor Dependency Injection

Spring Bean Configuration File :

```
<bean id="bankService" class="bank.server.internal.BankImpl">
  <constructor-arg ref="accountRepo"/>
  <constructor-arg ref="dbAccessService"/>

</bean>
<bean id="accountRepo" class="bank.server.internal.AccountRepositoryImpl">

</bean>
</bean>
<bean id=" dbAccessService " class="bank.server.internal.DBAccessServiceImpl">

</bean>
```

2 class instances are expected to be passed to constructor

Equivalent to

```
AccountRepository accountRepo         = new
    bank.server.internal.AccountRepositoryImpl();
DBAccessRService dbAccessService = new bank.server.internal.DBAccessServiceImpl();

Bank Bank = new BankImpl(accountRepo, dbAccessService);
```

Setter Dependency Injection

Spring Bean Configuration File :

```
<bean id="banService" class="bank.server.internal.BankImpl">
  <property name="accountRepository" ref="accountRepo"/>
  <property name="dBAccess" ref="dbAccessService"/>
</bean>
<bean id="accountRepo" class="bank.server.internal.AccountRepositoryImpl">
</bean>
</bean>
<bean id="dbAccessService" class="bank.server.internal.DBAccessServiceImpl">
</bean>
```

2 class instances are expected to be passed to Setters

Equivalent to

```
AccountRepository accountRepo = new bank.server.internal.AccountRepositoryImpl();
DBAccessRService dbAccessService = new bank.server.internal.DBAccessServiceImpl();

Bank Bank = new BankImpl();
bank.setAccountRepository(accountRepo);

bank. setDBAccess(dbAccessService);
```

Setter vis-à-vis constructor Dependency Injection

Constructor Dependency : When Do I Use

Enforce mandatory dependancy
Promote Immutability(not setters)

Setter Dependency : When do I Use

Allow optional dependencies
Allow defaults
Config file is more readable(property names are generally meaningful)
Inherited automatically(Constructors are required to be written in sub-classes)

Setter vis-à-vis constructor Dependency Injection

- **Mix-and –Match**

```
<bean id="banService" class="bank.server.internal.BankImpl">
   <constructor-arg ref="accountRepo"/>
  <property name="dbAccess"ref="dbAccessService"/>

</bean>
<bean id="accountRepo" class="bank.server.internal.AccountRepositoryImpl">

</bean>
</bean>
<bean id=" dbAccessService " class="bank.server.internal.DBAccessServiceImpl">

</bean>
```

Reccomendations

Use Constructor DI for required properties

Use Setter Injection for optional properties and defaults.

Be consistent in the approach.

Mix-and –Match

```
<bean id="banService" class="bank.server.internal.BankImpl">
   <constructor-arg ref="accountRepo"/>
   <property name="dbAccess"ref="dbAccessService"/>

</bean>
<bean id="accountRepo" class="bank.server.internal.AccountRepositoryImpl">

</bean>
</bean>
<bean id=" dbAccessService " class="bank.server.internal.DBAccessServiceImpl">

</bean>
```

Installing and Using STS

Lab -2 : 40 minutes

Use STS for creating Spring App

Injection for scalars

- <bean id="*someService*" class="*example.SomeServiceImpl*">
- <property name="balance" value="*10000*"/>
- </bean>

- Container takes care of auto conversion of scalar values .
- Throws Exception if type mis-match (non-convertible types)

Injecting Collections

```
<bean id="bankService" class="bank.server.BankImpl">
    <property name="accounts">
    <list>
        <ref bean ="current" >
        <ref bean ="savings" />
        <ref bean = "FD" />
        <ref bean ="housingLoan />
    </list>
</bean>
```

```
<set>
    <ref bean ="current">
    <ref bean ="savings"/>
    <ref bean = "FD /
    <ref bean ="housingLoan"/>
    <ref bean ="housingLoan"/> //duplicate entry
</set>
```

```
<bean id="current" class="AccountServiceImpl">
<bean id="savings" class="AccountServiceImpl">
<bean id="FD" class="AccountServiceImpl">
<bean id="housingLoan" class="AccountServiceImpl">
```

```
class BankImpl implements Bank{
    Collection<AccountService> accounts;
    BankImpl(Collection<AccountService> accounts){
        this. accounts = accounts;
    }
    public void showAccountServices(){
        for(AccountService acct : accounts){
            System.out.println(acct.toString());
        }
```

Injecting Collections

```
<bean id="bankService" class="bank.server.BankImpl">
    <property name="accounts">
    <map>
       <entry  key="CurrentAccount"   value-ref="current" >
       <entry  key="SaningsAccount"   value-ref  ="savings" />
       <entry  key="FixedDeposit"      value-ref = "FD"  />
</map>

</bean>
```

```
<map>
    <entry key-ref="someBean"
           value-ref "otherbean"/>

</map >
OR
<map>
  <entry key="someScalarType"
         value="someScalarType"/>

</map>
```

```
class  BankImpl  implements   Bank{
   Map<String ,AccountService>  accounts;
   BankImpl(Map<String, AccountService> accounts){
       this. accounts =  accounts;
   }
   public void  showAccountServices(){
      for(String accountType : accounts.keySet()){

          System.out.println(accountServices.get(acctType).toString());

      }

}
```

Injecting Collections : Properties

```
<bean id="bankService" class="bank.server.BankImpl">
    <constructor-arg name="accounts">
    <props>
       <prop  key="CurrentAccount">current  </prop>
       <prop key="SaningsAccount"  >savings</prop>
       <prop  key="FixedDeposit" >FD</prop>
</props>

</bean>
```

```
<props>
 <prop key="keyString"
>value</prop>


</props>
```

```
 class  BankImpl  implements   Bank{
    Properties<String, String>  account;
    BankImpl(Properties accounts){
        this. accountServices =  accounts;
    }
    public void  showAccountServices(){
       for(String accountType : accountServices.keySet()){

           System.out.println(accountServices.get(acctType).toString());

       }

}
```

DI Advance Options:Inner-beans

```
<bean id="bankService" class="bank.server.internal.BankImpl" abstract="true" >
  <constructor-arg >
    <bean class="bank.server.internal.AccountRepositoryImpl" />
  </constructor-arg>
  </bean>
```

Implications :

1. Inner beans are always lazy initialized as opposed to early-init of outer beans
2. Inner beans are always scoped to scope of the outer bean

Installing and Using STS

Lab -3 : 40 minutes

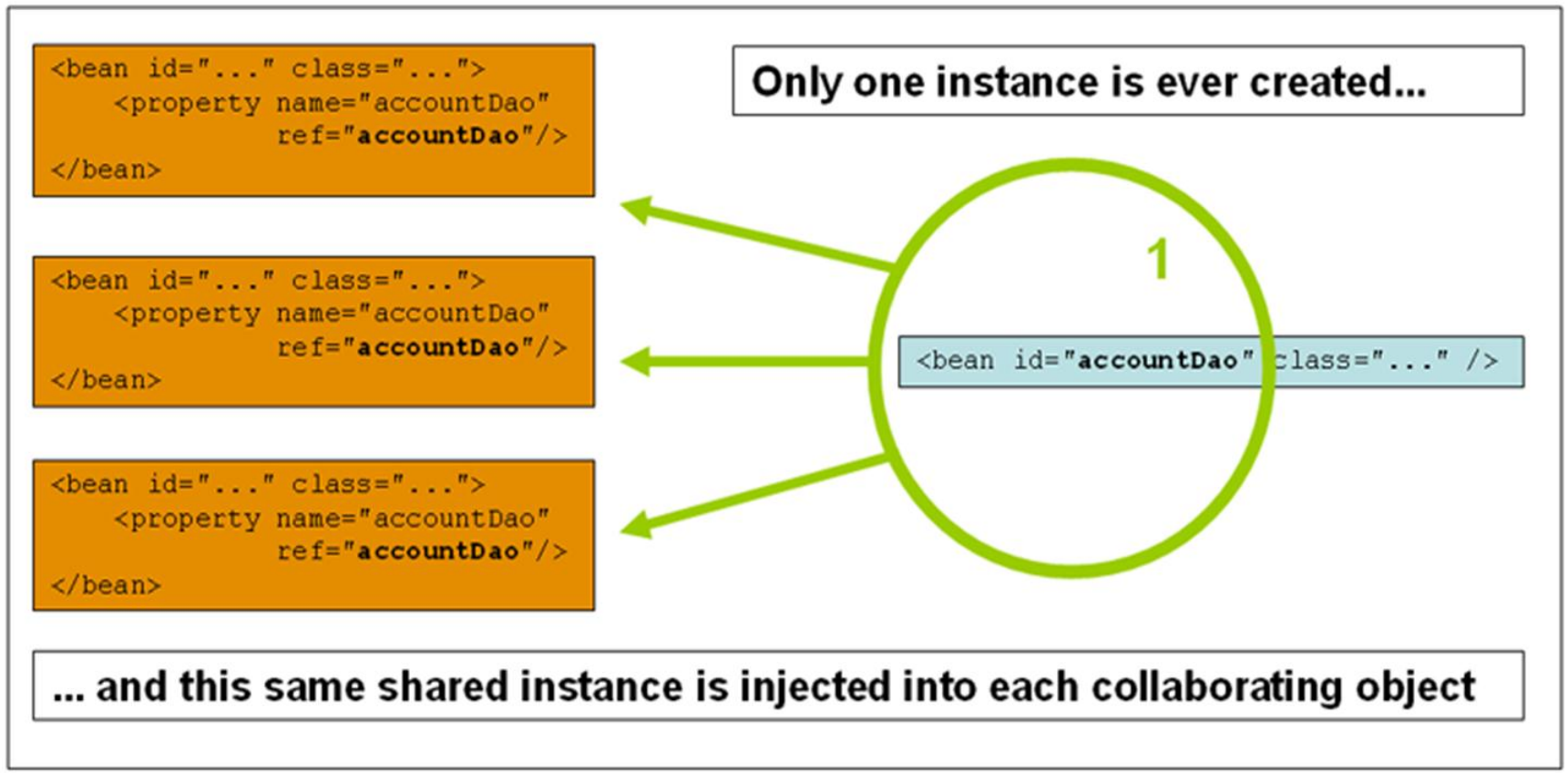
Use DI for collections

Controlling Bean Creation

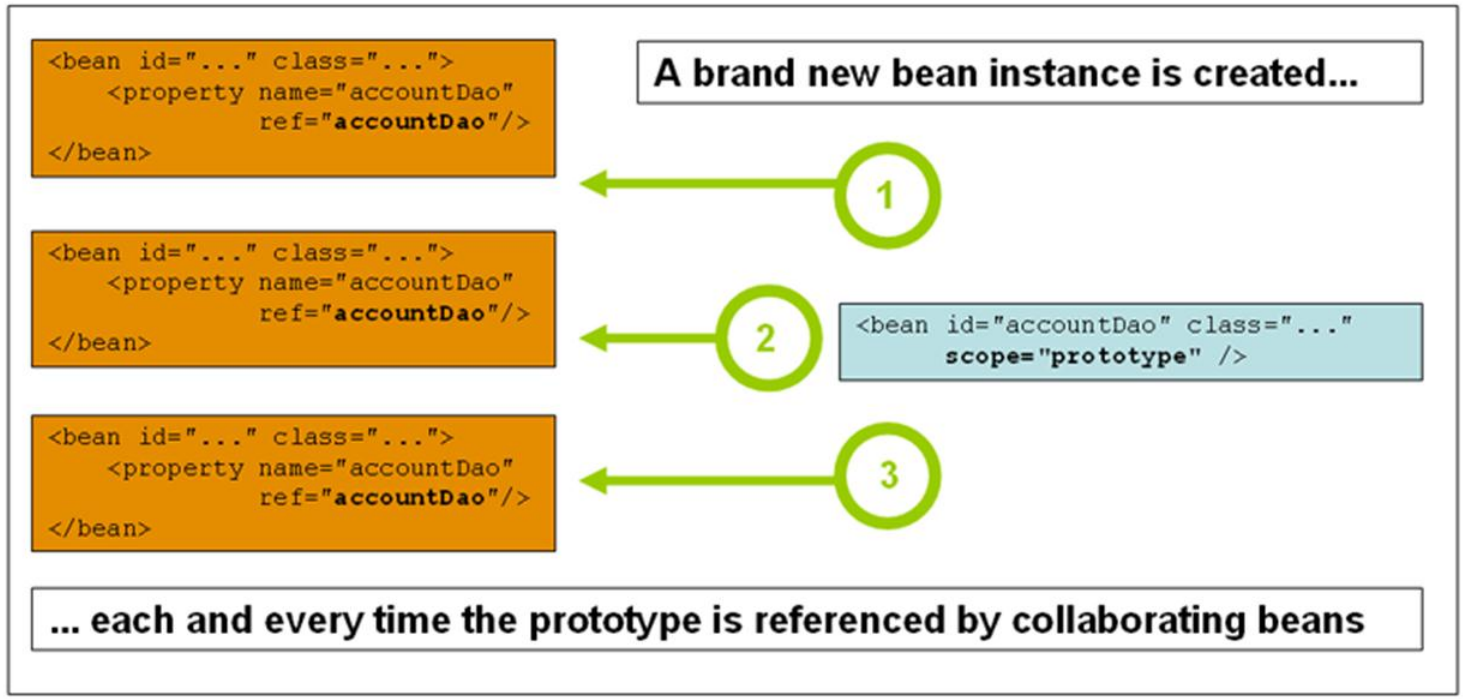
Bean Scopes

- Control number of instances of a specific bean :
  - "**singleton** : 1 Per application (this is default)
  - **prototype** : 1 Per usage/per call to getBean() (scope"prototype")(This is useful if domain classes are used as beans)
  - **request** : 1 Per user-request (If HTTP capable Spring context is used using SpringMVC then scope="request")
  - **session** : 1 per session (If HTTP capable Spring context is used using SpringMVC then scope="session")

Bean Scopes : singleton(default)



Bean Scopes : prototype



Bean Scopes : prototype

Destruction of prototypes is NOT managed by container .

You have to do the clean-up(call destroy method etc) explicitly in your client code

You need to type-cast ctx to ConfigurableApplicationContext and call close()

OR create custom-BeanPostProcessor to do it for you.

Prototypes are not eager-initialized unlike singletons

Prototypes are injected as one instance per singleton dependants.

DI Advance Options:sub-beans

- Get same bean class with different id's and with some common DI.
- Over ride few property values.
- Minimize repetitive declarations in xml
- parent attribute for sub-beans
- No class attribute
- abstract="true" for parent bean

```
<bean id="bankService" class="bank.server.internal.BankImpl" abstract="true" scope="singleton" >
  <constructor-arg >
    <bean class="bank.server.internal.AccountRepositoryImpl" scope="singleton"/>
  </constructor-arg>
  <constructor-arg>
    <set>
      <ref bean="current" />
      <ref bean="saving" />
      <ref bean="loan"/>
    </set>
  </constructor-arg>
  <property name="bankName" value="RBI" />
  <property name="branchCode" value="1234" />
</bean>
<bean id="sbi" parent="bankService" >
<property name="bankName" value="StateBank" />
</bean>
<bean id="axis" parent="bankService" >
<property name="branchCode" value="345" />
</bean>
```

DI Advance Options:sub-beans with only property inheritance

- Parent bean could be an abstract class an interface, a concrete class or may not be a class at all(it is just a set of properties that child beans share).
- abstract="true" for parent bean
- Child beans are actually classes which are un-related classes sharing common properties.
- Child beans can overwrite the property values.

```
<bean id="bankService" abstract="true" >
  <constructor-arg >
    <bean class="bank.server.internal.AccountRepositoryImpl" />
  </constructor-arg>
  <constructor-arg>
    <props>
      <prop key="CURRENT ACCOUNT" >hello </prop>
      <prop key="SAVING ACCOUNT">saving</prop>
      <prop key="LOAN ACCOUNT" >loan </prop>
    </props>
  </constructor-arg>
  <property name="branchCode" value="1234" />
  <property name="bankName" value="SBI"></property>

</bean>
  <bean id="sbi" class="bank.server.internal.SBI" parent="bankService" >
  </bean>
  <bean id="axis" class="bank.server.internal.AXIS" parent="bankService" >
    <property name="branchCode" value="345" />
    <property name="bankName" value="AXIS"></property>
  </bean>
</bean>
```

Installing and Using STS

Lab -4 : 40 minutes

More DI Options : sub-beans, bean scopes

Controlling Bean Creation
I. Using Factory Method

- Special attributes : factory-method/init-method/destroy-method

- l
  - <bean id="someId" class="someClass" factory-method="getInstance() />

- A static object creation method can be invoked instead of calling constructor. Works for objects with private constructors and true singletons.

- <bean id="someBean" class="someClass" init-method="doInit" />
- <bean id="someBean" class="someClass" destroy-method="doDestroy" />

Controlling Bean Creation
II. Using factory-bean

- Special attributes : factory-bean Using POJO factory
- Useful for existing factory classes

- <bean id="someId" class="someClass" factory-bean="someFactory" factory-method="getInstance" />
- <bean id="someFactory" class="someFactoryClass" />

An instance object creation method can be invoked instead of calling constructor. Works for objects with private constructors and true singletons

Controlling Bean Creation
III. Using FactoryBean implementation

- Use FactoryBean implementation
  - <bean id="someFactoryId" class="SomeFactoryClass" />
  - <bean id="bankService" class="BankImpl" >
  - <property name="service" ref="someFactoryId" />
  - </bean>

- A Dependency Injection of the Object returned by the Template Factory is done transparently (calling getObject Method)
- e.g.
- Class SomeFactoryClass implements FactoryBean<SomeService>{
- public SomeService getObject() throws Exception { return new SomeSevice() ;}

Injecting Methods

- Runtime Method injection
- 2 Forms
  - **SpringBean Replacer :** Returns a different spring bean at runtime
  - **Method Replacer :** Runtime injection of different implementation

SpringBean replacer :

•Bean class could be abstract(or concrete)

•abstract class Magician {

• abstract MagicBox getBox();

•}

•Actual method implementation done in a class which is a subclass of Magician To replace method, subclass is created and instantiated at run time by CGLIB

•Byte Code is instrumented by CGLIB at runtime.The method is implemented into it.

•So class who's method is to be replaced can not be final, method cannot be final either.

Configuration :

```
<bean id="magician" class="com.rits.Magician" >
            <property name="magicwords" value="Choo Mantar
!!"></property>
            <lookup-method name="getBox" bean="beauty"
></lookup-method>
</bean>
  <bean id="magician2" parent="magician">
    <lookup-method name="getBox" bean="beauty" />
  </bean>
  <bean id="elephantbox" class="com.rits.ElephantBox" />
  <bean id="beauty" class="com.rits.BeautifulAssistantBox" />
```

Method Replacer

```
<bean id="magician" class="com.rits.Magician" >
        <property name="magicwords" value="Choo Mantar
!!"></property>
        <replaced-method name="getBox"
bean="elephantReplacer" ></lookup-method>
</bean>
  <bean id="magician2" parent="magician">
    <replaced-method name="getBox" bean="beautyReplacer" />
  </bean>
  <bean id="elephantReplacer" class="ElephantReplacer" />
  <bean id="beautyReplacer" class="BeautifulAssistantReplacer" />
```

Method Injection using Method-Replacer :

```
public class ElephantReplacer implements MethodReplacer {

        public Object reimplement(Object arg0, Method arg1, Object[] arg2)
                        throws Throwable {
                                //System.out.println("Object whos method is to be
replaced"+arg0.toString()+" Name of Method "+arg1.getName()+" MYSORE ELEPHANT "+arg2[0]);
                                return new ElephantBox();

        }

}
```

Injecting Methods

```
<bean id="magician" class="com.rits.Magician" >
            <property name="box" ref="magicbox"></property>
            <property name="magicwords" value="Choo Mantar
!!"></property>
            <replaced-method  name="perform"
replacer="performReplacer">
              <arg-type >com.rits.MagicBox</arg-type>
              <arg-type >String</arg-type>
            </replaced-method>
          </bean>
   <bean id="magicbox" class="com.rits.MagicBoxImpl" >
     <replaced-method name="showContents"
replacer="elephantReplacer" >
       <arg-type>String</arg-type>
     </replaced-method>
     </bean>
   <bean id="elephantReplacer" class="com.rits.ElephantReplacer" />
   <bean id="performReplacer"
class="com.rits.PerformMethodReplacer" ></bean>
```

Injecting Non-spring beans

- Using non-managed beans (User instantiated beans) with Spring DI
- These are normally Domain Beans(Pojos) created by say hibernate session.load(). You want these objects to be monitored for dependency etc. by Spring.
- Use VM args to enable Load Time Weaving of AspectJ's Aspect
- -javaagent:C:\core-spring-3.1.0.RELEASE\repository\org\aspectj\aspectjweaver\1.6.8\aspectjweaver-1.6.8.jar

```
import org.springframework.beans.factory.annotation.Configurable;
@Configurable("magician")
public class Magician {
String magicwords;
MagicBox box;
…..}
<context:spring-configured />
<bean id="magician" class="com.rits.Magician" >
   <property name="box" ref="magicbox"></property>
   <property name="magicwords" value="Choo Mantar !!" />
</bean>
   <bean id="magicbox" class="com.rits.MagicBoxImpl" />
```

AutoWiring

- <bean id="*bankService" class="bank.server.BankImpl*">
- <property name="accountService" *ref="acctRepo"* />
- </bean>
- <bean id="*acctRepo" class="bank.server.internal.AccountRepositoryImpl*" />
- OR
- <bean id="*bankService" class="bank.server.BankImpl" autowire="byName"*/>
- <bean id="*acctRepo" class="bank.server.internal.AccountRepositoryImpl*" />

- Autowiring of 4 types

- 1. **byName** : Attempts to find a bean in the container whose name(or ID) is the same as the name of the property being wired(injected)
- 2. **byType** : Attempts to find a single bean in the container whose type matches the type of the property being wired. If not found it ignores, if multiple macthes found then throws org.springframework.beans.factory.UnsatisfiedDependencyException
- 3.**constructor** : Tries to match up one or more beans in the container with the parameters of one of the constructors of the bean being wired. In case of ambiguity it throws org.springframework.beans.factory.UnsatisfiedDependencyException
- 4. **autodetect :** Tries to apply constructor stretagy first if matching constructor found then it will use contructor injection if not then will apply byType strategy and is matching type found then applies setterInjection if neither is found then leaves property un-initialized.

FactoryBean

- Spring uses FactoryBean Template interface

  RemoteProxies are created using FactoryBean

  – DataSources are created using FactoryBean for accessing Hibernate, iBatis etc.

ApplicationContext

- Spring provides a lightweight container called “ApplicationContext”
  - 3 flavours for resource(config file) loading

  - 1. new ClassPathXmlApplicationContext(“com/sungard/application-config.xml”);
  - $CLASSPATH/com/sungard/application-config.xml”

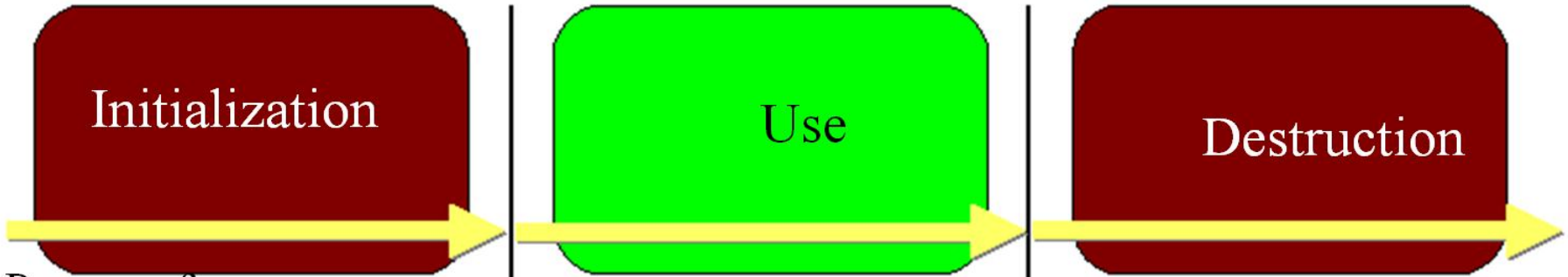
  - **2.** new FileSystemXmlApplicationContext(c:\\somedir\\application-config.xml”);
  - OR
  - new FileSystemXmlApplicationContext(../application-config.xml”);

  - **3.** XmlWebApplicationContext : Created by WebCOntainers resource loading happens relative to WEB-INF/

ApplicationContext

- Loading multiple config files
  - ApplicationContext ctx =
  - new ClassPathXmlApplicationConext(new String[]{“application-config.xml”, “oracle-infra-config.xml” , file:../someOther-config.xml”});
- Separate files for Infrastructure config than actual application beans.

Phases of the Application Lifecycle



- Prepares for use
- Application services
  - Are created
  - Configured
  - May allocate system resources
- Application is not usable until this phase is complete

- Used by clients
- Application services
  - Process client requests
  - Carry out application behaviors
- 99.99% of the time is spent in this phase

- Shuts down
- Application services
  - Release any system resources
  - Are eligible for garbage collection

Application Initialization

When We create an instance of the ApplicationContext :

Initialization is Complete

During applicationContext Creation, What Happens?

(when we call new CLassPathXMLApplicationContext(…)

1. Loading of Bean Definitions from config files
2. Creates dependancy graph
3. All singleton beans and pre-instantiated (unless lazy-init="true")
4. Initialization of bean Instances

Application Initialization :

Loading Bean Definitions

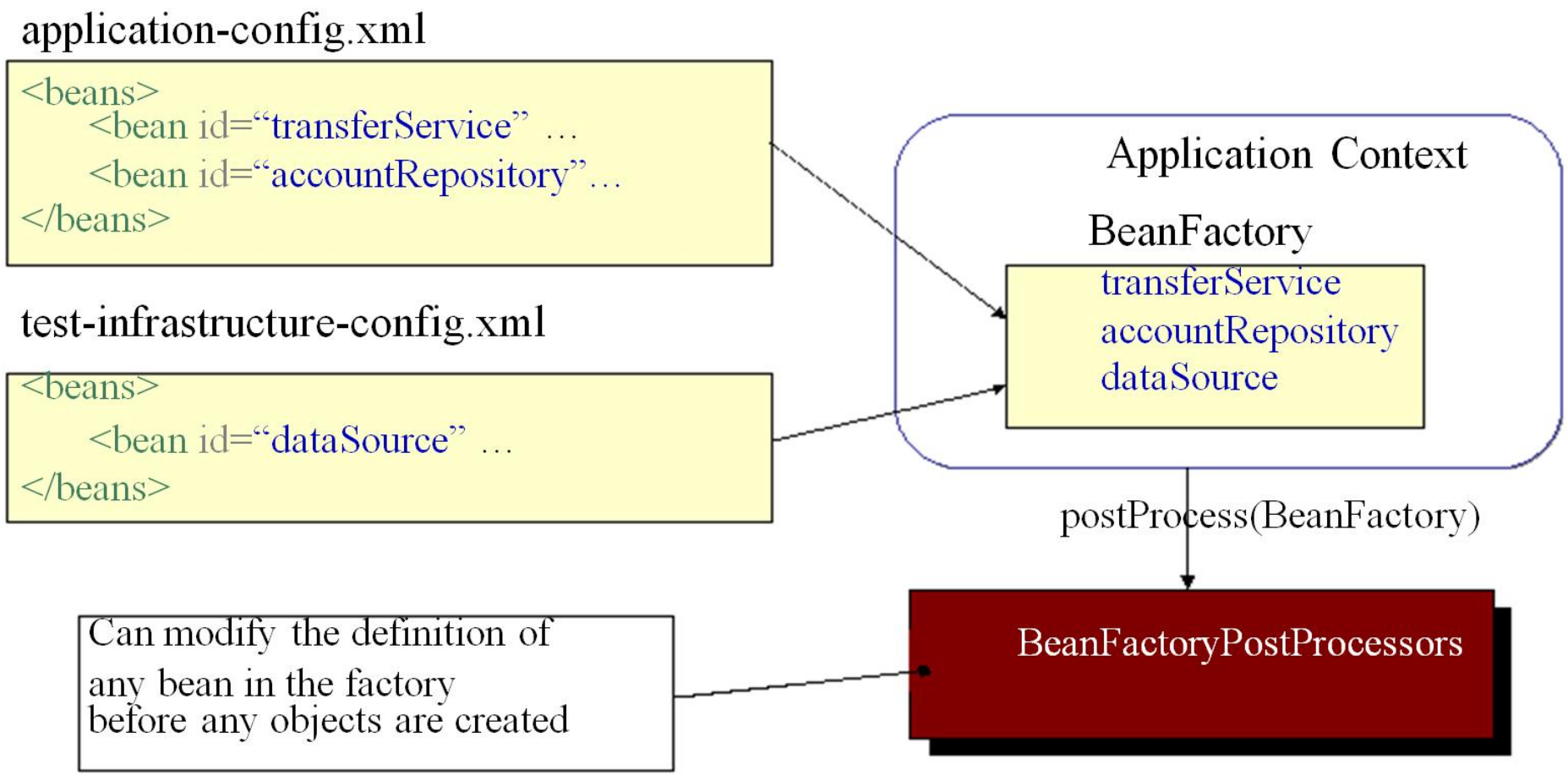
- The XML files are parsed.

Bean Definitions are loaded into the context's BeanFactory

Special Beans "BeanFactoryPostProcessor" are invoked

These special beans can modify the bean definitions before creating the objects.

Load Bean Definitions



BeanFactoryPostProcessor

•An Extension Point to modify bean definitions programatically before loading bean definitions

•Several useful special BeanFactoryPostProcessor beans are provided by Spring

- e.g. PropertyPlaceholderConfigurer
  It substitutes ${variables} in bean definitions with values from .properties files
  CustomEditorConfigurer
  It loads custom editors into beanfactory.

You can write your own by implementing BeanFactoryPostProcessor interface

Custom Property Editors


```
class Contact{
public PhoneNumber getPhone() {
return phone;
}

public void setPhone(PhoneNumber phone) {
this.phone = phone;
}
```

```
<bean id="contact" class="com.rits.training.Contact">
    <property name="phone" value="512-230-9630" />
</bean>
```

Need to convert String into PhoneNumber

```
public class PhoneNumber {
    String areaCode;
    String stdCode;
    String number;
public PhoneNumber(String areaCode, String stdCode, String number) {
     super();
     this.areaCode = areaCode;
     this.stdCode = stdCode;
     this.number = number;
}
```


```
mport java.beans.PropertyEditorSupport;

public class PhoneEditor extends PropertyEditorSupport {

@Override
public void setAsText(String text) throws IllegalArgumentException {
        String stripped = stripOutNonNumerics(text);
        String areaCode = stripped.substring(0,3);
        String prefix   = stripped.substring(3,6);
        String number   = stripped.substring(6);
        PhoneNumber phone = new PhoneNumber(areaCode,prefix, number);
        this.setValue(phone);
}
private String stripOutNonNumerics(String original){
        StringBuffer allNumeric = new StringBuffer();
        for(int i=0; i < original.length(); i++){
        char c = original.charAt(i);
        if(Character.isDigit(c)){
        allNumeric.append(c);
}
```

We should tell Spring about our custom PhoneEditor

```
<bean id="contact" class="com.rits.training.Contact" >
    <property name="phone"  value="512-230-9630" />
</bean>

<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="com.rits.training.PhoneNumber"
                    value="com.rits.training.PhoneEditor" />
            </entry>
        </map>
    </property>
</bean>
```

CustomEditorConfigurer is a BeanFactoryPostProcessor. It registers PhoneEditor with BeanFactory, so BeanFactory would invoke this before creating Contact bean.

BeanFactoryPostProcessor : PropertyPlaceholderConfigurer

```
<beans>
   <bean id="dataSource" class="com.oracle.Jdbc.pool.OracleDataSource">
            <property name="URL" value="${datasource.url}" />
            <property name="user" value="${datasource.user}" />
    </bean>
<bean class="org.springframework.beans.factory.config.
                PropertyPlaceholderConfigurer">
   <property name="location" value="/WEB-INF/datasource.properties" />
</beans>
```

Variables to replace

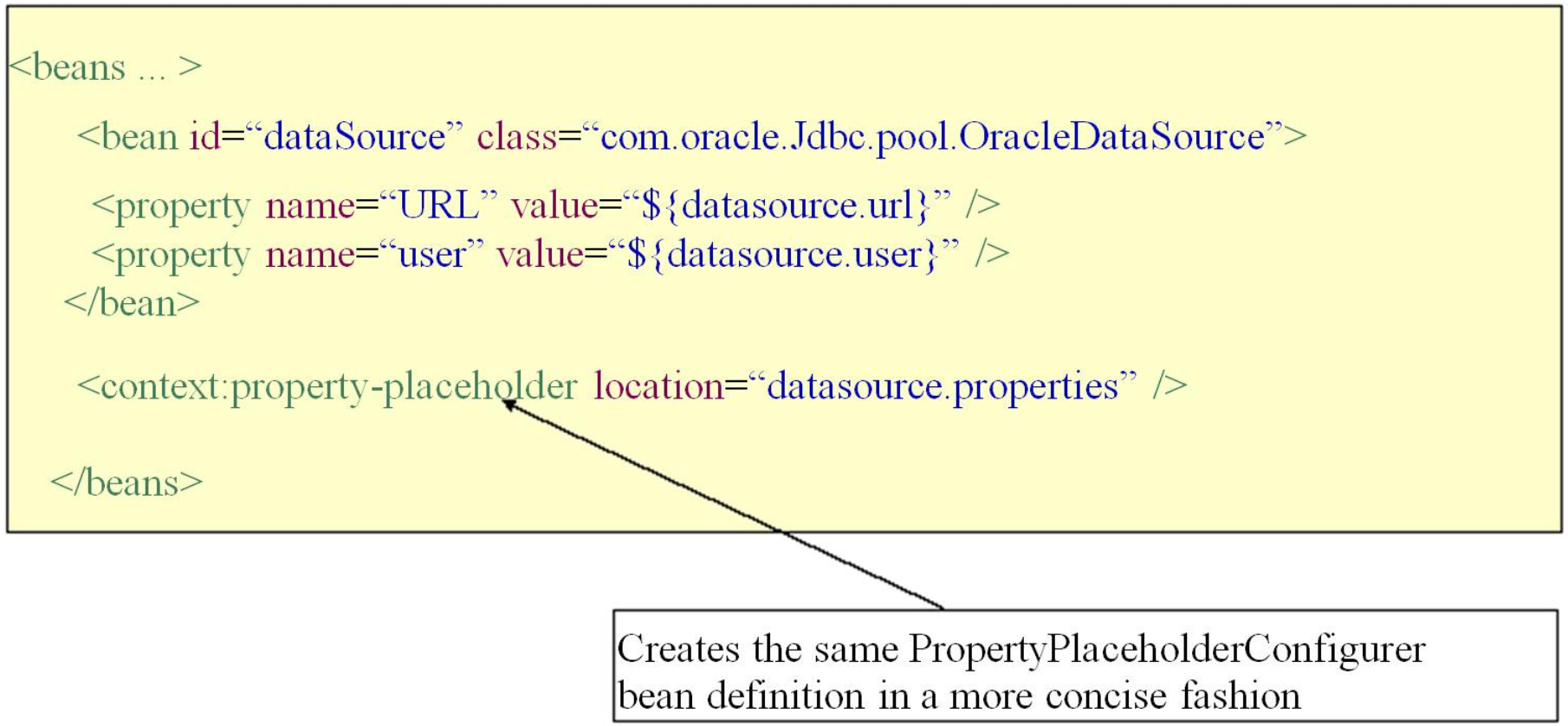
File where the variable values reside

datasource.properties

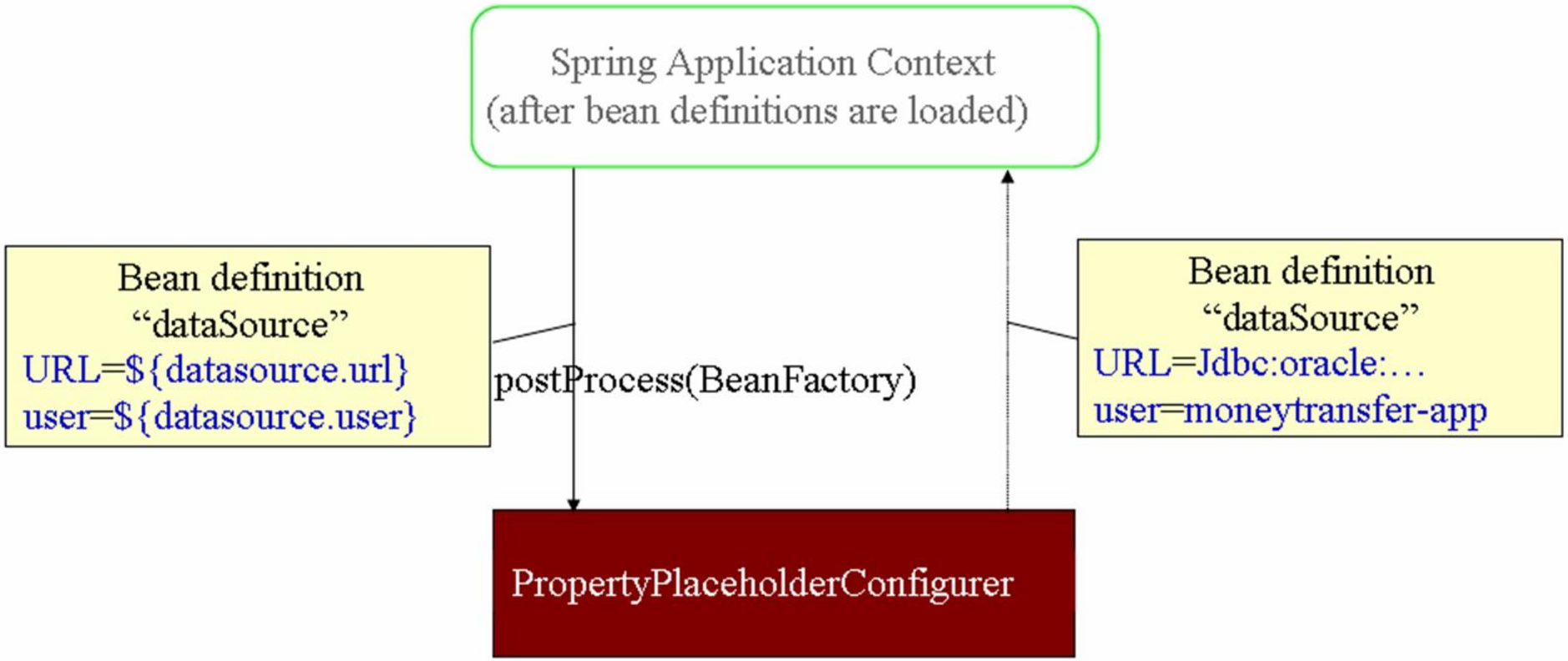
```
datasource.url=jdbc:oracle:thin:@localhost:1521:BANK
datasource.user=moneytransfer-app
```

Easily editable

Simplifying configuration with <context:property-placeholder>



How PropertyPlaceholderConfigurer Works



Spring Expressions SpEL

Evaluating dynamically some values and assigning them to properties

syntax : #{ expression }

e.g. #{ T(java.lang.Math).random() }

- **Boolean and relational operators :**
- **person.age==23**
- **Standard Expressions :**
- **student.address.city**
- **Class Expression : T() : This tells container to act on the Class object and not on instance of class**

T(java.lang.Math).random()

Spring Expressions SpEL

Accessing arrays, lists and Maps

T(java.util.Arrays).asList('a','b','c','d')[1] This will return b

<someBeanId>.someMap['key']

Method Invocations

'Hello World'.toLowerCase()

Calling Constructors

new Shape("Circle");

Ternary operators

T(java.lang.Math).random() > 0.5 ? "I win Lottery ": "I loose money"

Built-in systemProperties expression

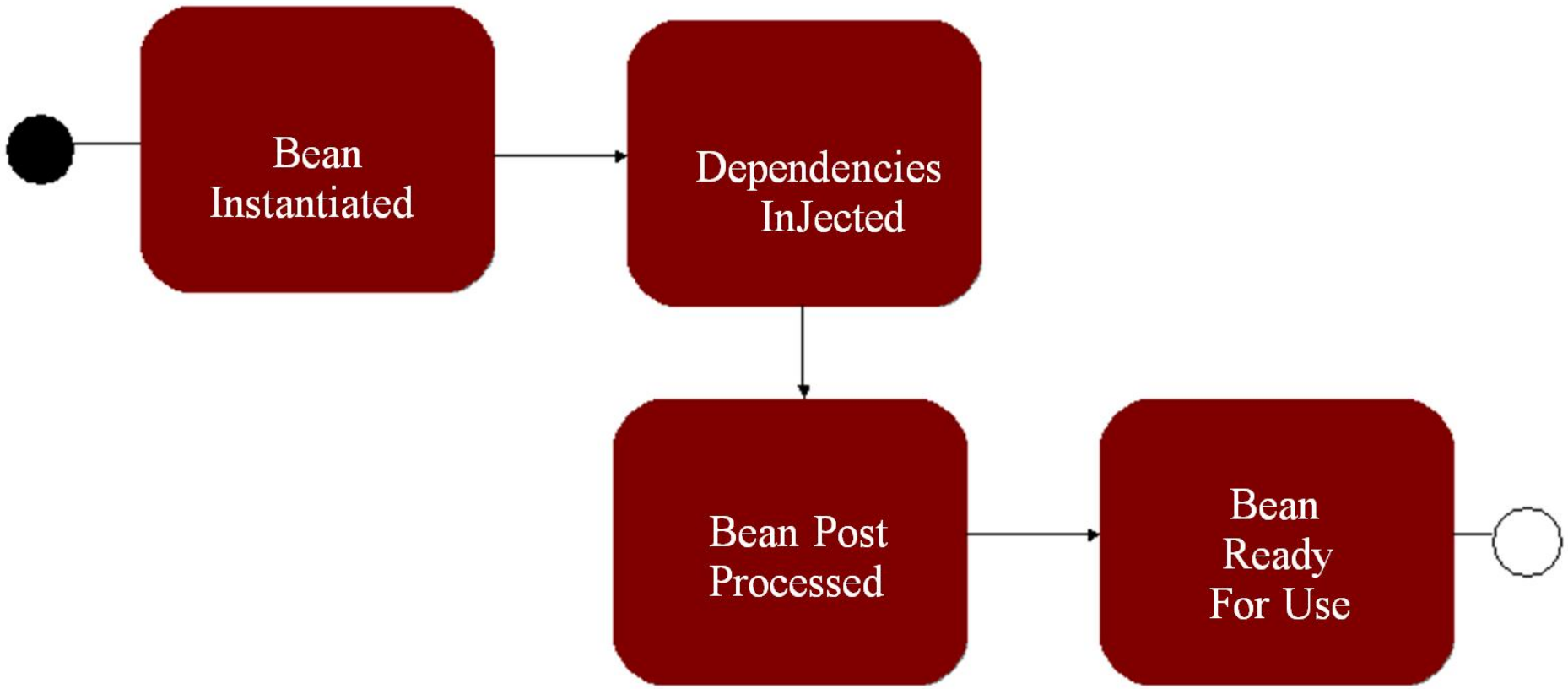
#{ systemProperties } : This will return Instance of Properties initialized by method System.getProperties()

#{ systemProperties['user.country'] }

Steps involved in Initializing Bean Instances

- Each bean is eagerly instantiated by default
  - Created in the right order with its dependencies injected
- After dependency injection each bean goes through a post-processing phase
  - Where further configuration and initialization can occur
- After post processing the bean is fully initialized and ready for use
  - Tracked by its id until the context is destroyed

Bean Initialization Steps



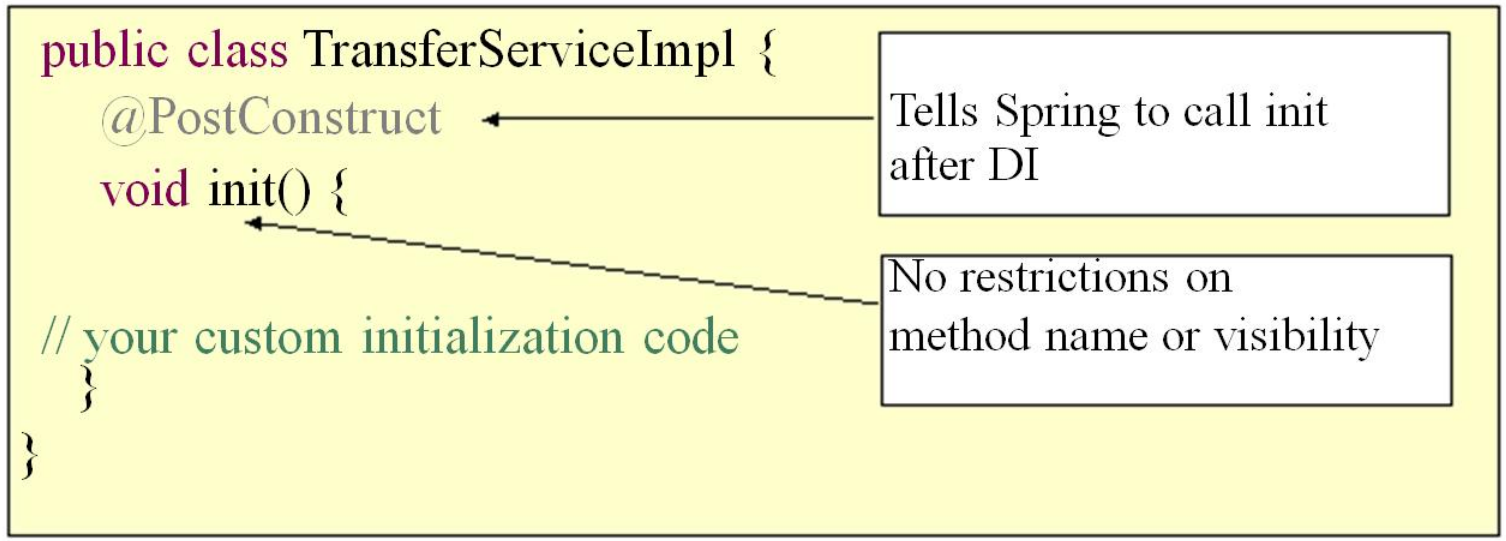
Bean Post Processing

- Bean post processing can be broken down into two steps
  - Initialize the bean if instructed
  - Call special BeanPostProcessors to perform additional configuration

Initialize the Bean if Instructed

- A bean can optionally register for one or more initialization callbacks
  - Useful for executing custom initialization behaviors
- There's more than one way to do it
  - JSR-250 @PostConstruct
  - <bean/> init-method attribute
  - Implement Spring's InitializingBean interface

Registering for Initialization with @PostConstruct



- This is the preferred initialization technique
  - Flexible, avoids depending on Spring APIs
  - Requires Spring 2.5 or better

Enabling Common Annotation- Based Initialization

- @PostConstruct will be ignored unless Spring is instructed to detect and process it

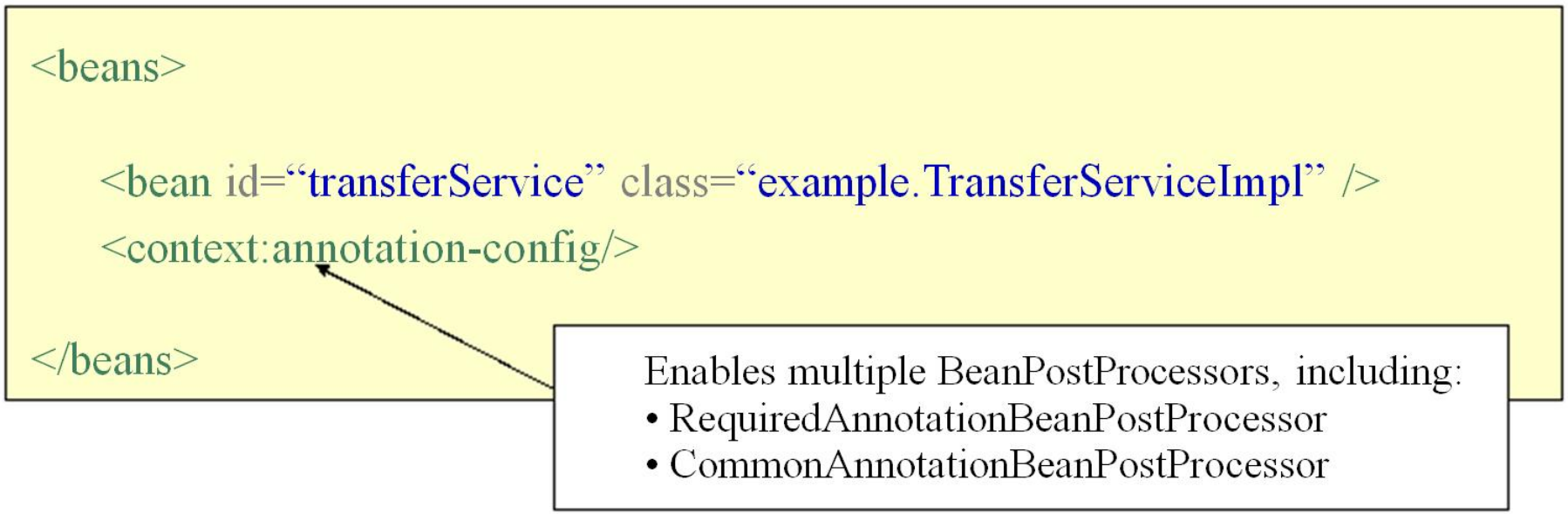
```
<beans>

    <bean id="transferService" class="example.TransferServiceImpl" />

    <bean class="org.springframework.context.annotation.
                        CommonAnnotationBeanPostProcessor" />
</beans>
```

Detects and invokes any methods annotated with @PostConstruct

Simplifying Configuration with <context:annotation-config>



Initializing a Bean You Do Not Control

Use init-method to initialize a bean you do not control the implementation of

```
<bean id="current"
class="bank.server.internal.CurentAccount"
init-method="initialize">
    ...
</bean>
```

Class with no Spring dependency

Method can be any visibility, but must have no arguments

Registering for an Initialization Callback with InitializingBean

- Initialization technique used prior to Spring 2.5
  - Disadvantage: Ties your code to the framework
  - Advantage: The only way to *enforce* initialization code should run (not affected by configuration changes)

```
package org.springframework.beans.factory;
    public interface InitializingBean {
        public void afterPropertiesSet();
}
```

Initialization : @Required

- RequiredAnnotationBeanPostProcessor

e.g.

```
@Required
public void setInterestRate(float interestRate) {
this.interestRate = interestRate;
System.out.println("Saving Account : interest ");
}rate property set");
```

If interestRate property of this class is not configured in config file for DI, container would throw

org.springframework.beans.factory.BeanCreationException

This is a useful technique for enforcing the initialization

Destroying : @PreDestroy

- BeanPostProcessor

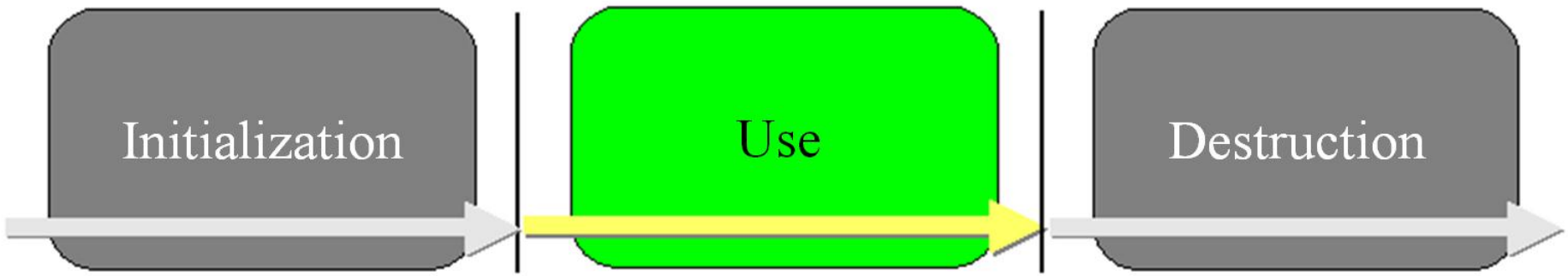
Called when ctx.close() is invoked.

```
@PreDestroy
public void releaseResources() {
    System.out.println("Resources released");
}
```

This is a useful technique for enforcing the release of some resources like DB connections etc.

Topics in this session

- Introduction to Spring's XML namespaces
- The lifecycle of a Spring application context
  - The initialization phase
  - **The use phase**
  - The destruction phase
- Bean scoping



The Lifecycle of a Spring Application Context (2) - The Use Phase

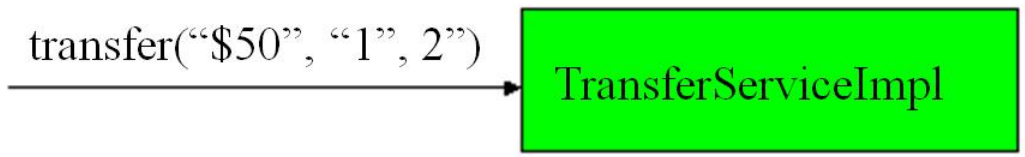
- When you invoke a bean obtained from the context the application is used

```
ApplicationContext context = // get it from somewhere
// Lookup the entry point into the application
TransferService service =
    (TransferService) context.getBean("transferService");
// Use it!
service.transfer(new MonetaryAmount("50.00"), "1", "2");
```

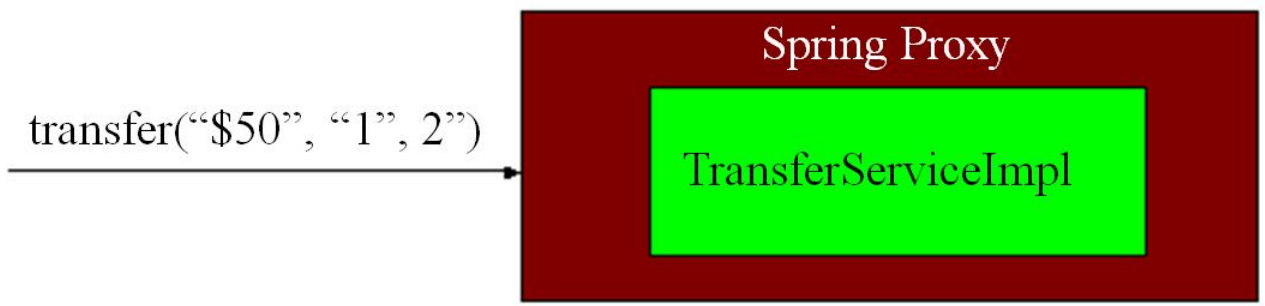
- But exactly what happens in this phase?

Inside The Bean Request (Use) Lifecycle

- If the bean is just your raw object it is simply invoked directly (nothing special)



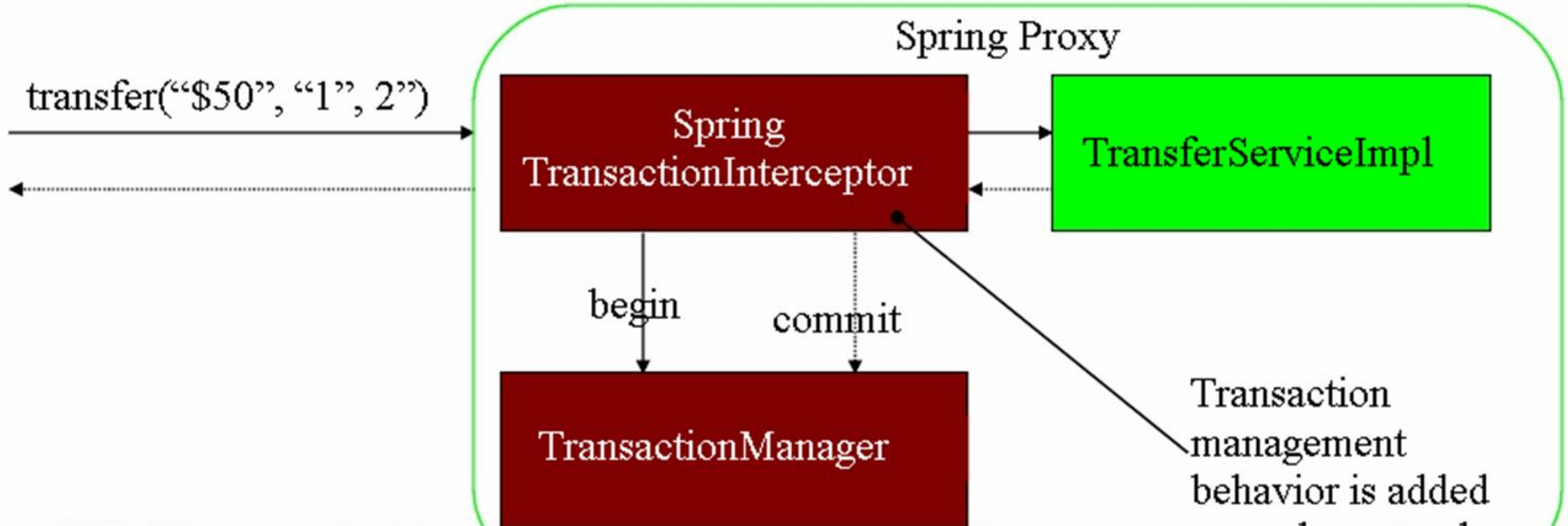
- If your bean has been wrapped in a proxy things get more interesting



How PropertyPlaceholderConfigurer Works

Proxy Power

- A BeanPostProcessor may wrap your beans in a dynamic proxy and add behavior to your application logic *transparently*



Annotation-Based Dependency Injection

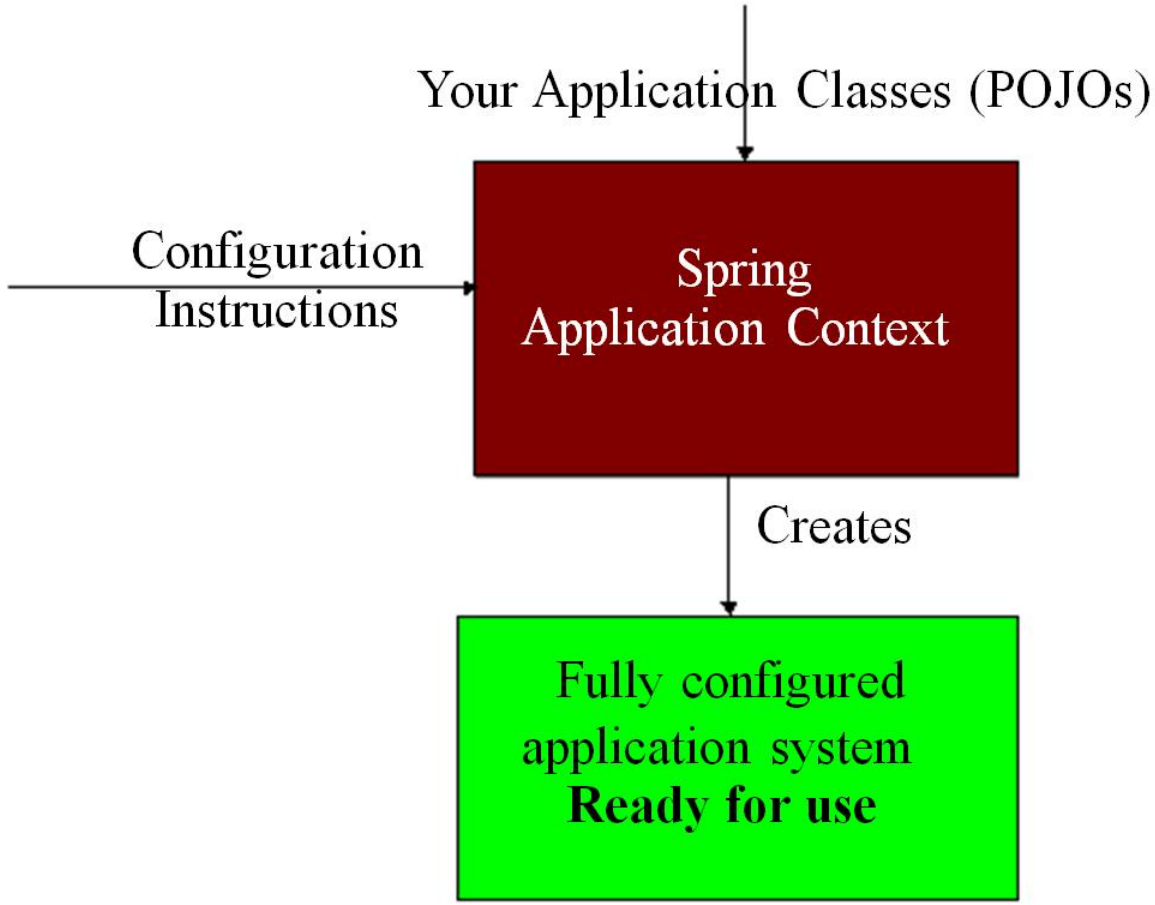
Introducing Spring's Annotations for

Dependency Injection

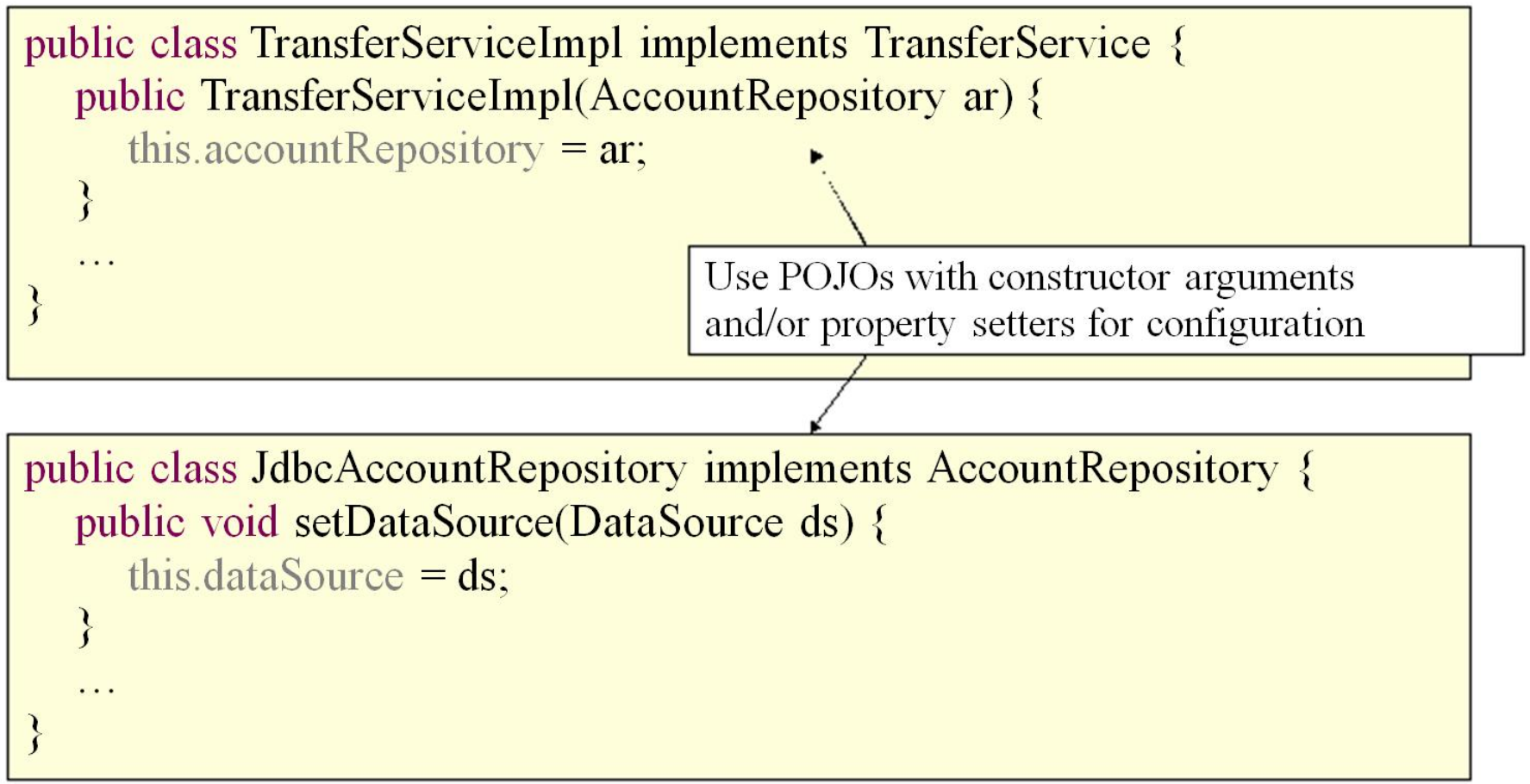
Topics in this Session

- Configuration Approaches
- Java-based Configuration
- Annotation Quick Start
- Other Configuration Annotations
- Component Scanning
- Stereotype and Meta-annotations
- When To Use What
- Summary & Lab
- Advanced Options

How Spring Works



The Approach You're Used to



The Approach You're Used to

<bean> *defines* a bean

```
<beans>

   <bean id="transferService" class="app.impl.TransferServiceImpl"">
      <constructor-arg ref="accountRepository" />
   </bean>

   <bean id="accountRepository" class="app.impl.JdbcAccountRepository"">
      <property name="dataSource" ref="dataSource" />
   </bean>

   <bean id="dataSource" class="com.oracle.jdbc.pool.OracleDataSource"">
      <property name="URL" value="jdbc:oracle:thin:@localhost:1521:BANK" />
      <property name="user" value="moneytransfer-app" />
   </bean>

</beans>
```

Alternative Approaches

- Bean definitions can also be defined and configured in **Java**
- Or by using **annotations** in Spring-managed classes
- Both are alternatives to XML approach
  - Not intended as complete replacements
  - Can be used together

Topics in this Session

- Configuration Approaches
- **Java-based Configuration**
- Annotation Quick Start
- Other Configuration Annotations
- Component Scanning
- Stereotype and Meta-annotations
- When To Use What
- Summary & Lab
- Advanced Options

Java-based Approach

- Spring 3.0 introduces Java-based configuration
  - Based on the old JavaConfig project
  - Configuration still external to POJO classes to be configured
- Use Java code to define and configure beans
  - More control over bean instantiation and configuration
  - More type-safety
- Still produces bean definitions
  - Spring remains in charge applying post-processors etc.

Spring3.0 Java Configuration Class

•The central artifact in Spring's new Java-configuration support is the @Configuration-annotated class.

•These classes consist principally of @Bean-annotated methods

•These Methods define instantiation, configuration, and initialization logic for objects to be managed by the Spring IoC container.

•Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions.

```
@Configuration
public class AppConfig {
      @Bean
      public MyService myService() {
          return new MyServiceImpl();
      }
}
```

XML Way

```
<beans>
<bean id="myService" class="com.rits.training.MyServiceImpl"/>
</beans>
```

Spring3.0 Java Configuration Class

@Configuration-annotated classes are sub-classed at load time by CGLIB so

They should not be final

They should have no-arg non-private constructor

```
@Configuration
public class AppConfig {
      @Bean
      public MyService myService() {
          return new MyServiceImpi();
      }
}
```

Getting bean instances using **AnnotationConfigApplicationContext**.

```
Class client{
   public static void main(String[] args){
       AnnotationConfigApplicationContext ctx =
                                  new AnnotaionConfigApplicationContext(AppConfig.class)
   }
```

No XML

Just Use

@Configuration class Name

Spring3.0 Java Configuration Class

@Mix-Match of Java-Based and XML based configuration (@Configuration –centric)

```
@ImportResource("test-infrastructure-Java_Based_config.xml")
public class BankConfig {

    private @Value("#{jdbcProperties.oracleURL}") String url;
    private @Value("#{jdbcProperties.oracleUser}") String username;
    private @Value("#{jdbcProperties.oraclePassword}") String password;
    private @Value("#{jdbcProperties.oracleDriver}") String driver;

    .....
        public @Bean(scope="singleton") DriverManagerDataSource dataSource() {
        return new DriverManagerDataSource(url, username, password);
    }
```

testdb.properties

```
oracleDriver=oracle.jdbc.driver.OracleDriver
oracleURL=jdbc:oracle:thin:@localhost:1521:XE
oracleUser=system
oraclePassword=system
```

test-infrastructure-…xml

```
<util:properties id="jdbcProperties"
location="testdb.properties"/>
```


```
@Configuration("configBank")
@ImportResource("test-infrastructure-Java_Based_config.xml")
public class BankConfig {
            private @Value("#{jdbcProperties.oracleURL}") String url;
            private @Value("#{jdbcProperties.oracleUser}") String username;
          .....
    @Bean(name="bankService")
    @Scope("prototype")
   Bank  bankService(){
   Collection<AccountService> accts = new ArrayList<AccountService>();
            accts.add(currentAccount());
          accts.add(loan());
            accts.add(saving());
            return new BankImpl(accountRepository(),accts);
   }
   @Bean(name="acctRepo")
  AccountRepository accountRepository(){
            return new AccountRepositoryImpl(dataSource());
  }
   @Bean(name="current")
  AccountService currentAccount(){
  CurentAccount acct= new CurentAccount("Ramkrishna  IT",1000000);
                        acct.setAcctNo(123456789);
                        return acct;

}   public @Bean DriverManagerDataSource dataSource() {
                        DriverManagerDataSource dataSource = new DriverManagerDataSource(url, username,
password);
                        dataSource.setDriverClassName(driverClassName);
                        return dataSource;
            }
}
```

Enabling Java-based Approach

- Making this work requires two steps:
  - Define config class as Spring Bean
  - Add <context:annotation-config/>

```
<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="...">

    <context:annotation-config/>
```

Turns on checking of configuration annotations for beans declared in this context

Usage - Nothing Changes!


```
// Create the application from the configuration
ApplicationContext context =
      new ClassPathXmlApplicationContext("application-config.xml"");
// Look up the application service interface
Bank service = (Bank) context.getBean("bankService"");
// Use the application
```

Implementing @Bean Methods

- Configuration in Java: call any code you want
  - Not just constructors and property setters
- May not be private
- Method name becomes bean name
- Call @Bean method to refer to bean in same class
  - We'll show how to refer to external beans later
- Same defaults as XML: beans are singletons
  - Use @Scope annotation to change scope

…

```
@Bean @Scope("prototype")
public StatefulService service() {
```

@Bean Lifecycle

- Init- and destroy methods can be specified
  - Using attributes of the annotation
  - init really isn't required, can just call method yourself

```
@Bean(destroyMethod="close")
public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setUrl("jdbc:derby:rewardsdb");
    // can call init method here if you need to
    return dataSource;
}
```

Implementing @Configuration Classes

- Must have default constructor
- Multiple @Configuration classes may exist
- Can import each other using @Import
  - Equivalent to <import/> in XML
  - Means included classes don't have to be defined as Spring beans anymore

```
@Configuration
@Import({SecurityConfig.class, JmxConfig.class})
public class RewardsConfig {
    ...
}
```

Scanning and auto-wiring for @Configuration classes

- Are picked up by component scanning
  - No more need for XML bean definition or @Import
- Can use autowiring
  - Like any Spring-bean
  - Gives access to external beans from @Bean methods
- Can be autowired into another @Configuration
  - Allowing for external @Bean methods to be called
  - Type-safe, IDE-friendly alternative to by-name wiring

Combined Example

```
@Configuration
public class InfraConfig {
    @Bean public DataSource dataSource() {
        DataSource dataSource = ...;
        return dataSource;
    }
}
```

Import not needed if InfraConfig was component-scanned

```
@Configuration @Import(InfraConfig.class)
public class RewardsConfig {
    @Autowired InfraConfig infraConfig;

    @Bean AccountRepository accountRepository() {
        JdbcAccountRepository repository = new JdbcAccountRepository();
        repository.setDataSource(infraConfig.dataSource());
        return repository;
    }
}
```

Alternative is to simply autowire DataSource directly

Topics in this Session

- Configuration Approaches
- Java-based Configuration
- **Annotation Quick Start**
- Other Configuration Annotations
- Component Scanning
- Stereotype and Meta-annotations
- When To Use What
- Summary & Lab
- Advanced Options

Annotation-based Configuration

- Spring 2.5 introduced annotation-based configuration as another alternative to XML

- Moves configuration instructions into your code
  - Unlike XML or Java-based approaches

- Can be used in combination with the other approaches

Annotation injection is performed *before* XML injection, thus the latter configuration will override the former for properties wired through both approaches.

The Annotation-Based Approach


```
public class TransferServiceImpl implements TransferService {
                                        Spring automatically tries to wire this
 @Autowired
    public TransferServiceImpl(AccountRepository ar) {
        this.accountRepository = ar;
    }
}
        …
```

```
    public class JdbcAccountRepository implements AccountRepository {
        @Autowired
        public void setDataSource(DataSource ds) {
    …      this.dataSource = ds;
    }
}
```

The Annotation-Based Approach


```
<beans>

  <context:annotation-config/>

   <bean id="transferService"
            class="app.impl.TransferServiceImpl"/""/>

    <bean id="accountRepository"
            class="app.impl.JdbcAccountRepository""/>


   <bean id="dataSource" class="com.oracle.jdbc.pool.OracleDataSource"">
      <property name="URL" value="jdbc:oracle:thin:@localhost:1521:BANK" />
      <property name="user" value="moneytransfer-app" />
   </bean>

</beans>
```

No need to specify constructor-arg or property

Quick Start Summary

- Spring container needs POJOs and wiring instructions to assemble your application

- Wiring instructions can be provided in XML…
- …but also using Java with @Bean-methods
- … or @Autowired annotations…
- …or a combination

Topics in this Session

- Configuration Approaches
- Java-based Configuration
- Annotation Quick Start
- **Other Configuration Annotations**
- Component Scanning
- Stereotype and Meta-annotations
- When To Use What
- Summary & Lab
- Advanced Options

Configuration Annotations

- @Autowired
- @Qualifier / @Primary
- @Resource
- @Value
- @Component / @Repository
- Shown already:
  - @Configuration / @Bean plus @Scope, @Import, etc.
  - @PostConstruct / @PreDestroy / @Required

@Autowired

- Asks Spring to autowire fields, methods or constructors by using the type`
  - Requires unique match!
- Removes the need to specify constructor-arg or property elements in XML`

```
public class TransferServiceImpl implements TransferService

    @Autowired
    public TransferServiceImpl(AccountRepository ar) {
        this.accountRepository = ar;
    }
    …
}
```

@Autowired

- Methods can have any name
- Methods and constructors can have multiple arguments that will all be injected

```
public class MyServiceImpl implements MyService

    @Autowired
    public void initialize(SomeDependency sd, OtherDependency od) {
        this.someDependency = sd;
        this.otherDependency = od;
    }
    …
}
```

Disambiguation with @Autowired

- Disambiguation needed when multiple beans of same type exist, one of which must be injected
- Using @Qualifier you can inject beans by name

```
@Autowired
@Qualifier("primaryDataSource")
private DataSource dataSource;

@Autowired
void init(@Qualifier("primaryDataSource") DataSource primary,
          @Qualifier("altDataSource") DataSource alternative)
{ … }
```

Specify name of bean to inject

Option-I use qualifier child element in <bean> definition

```
<bean  class="com.rits.SomeDBSource">
    <qualifier value="primaryDataSource" >
</bean>

<bean  class="com.rits.SomeOtherDBSource">
    <qualifier value="altDataSource" >
</bean>
```

Disambiguation with @Primary

Option-II Java-based config alternative solution:

one bean can be made the primary candidate for autowiring

– Will be used in case of multiple matches by type

```
@Bean @Primary
public DataSource standardDataSource() {
    …
}

@Bean
public DataSource backupDataSource() {
    …
}
```

@Resource ("myBeanName")

- JSR-250 annotation (not Spring-specific)
- Injects a bean identified by the given name
- Useful when auto wiring by type does not work
- Can also lookup JNDI references

```
public class TransferServiceImpl implements TransferService {

    @Resource("myBackupRepository")
    public void setBackupRepo(AccountRepository repo) {
        this.backRepo = repo;
    }
}
```

@Resource in Spring

Some additional behavior over Java EE 5:

- By default, falls back to autowiring by type if no bean with given name exists
- Works for *all methods*, not just setters
  - But not for constructors!

SpEL and Annotations

- Use @Value for SpEL expressions in Java code
  - On a field or method or on parameter of autowired method or constructor

```
@Value("#{ systemProperties['user.region'] }")
private String defaultLocale;

@Value("#{ systemProperties['user.region'] }")
public void setDefaultLocale(String defaultLocale) { … }

@Autowired
public void configure(AccountRepository accountRepository,
      @Value("#{ systemProperties['user.region'] }") String defaultLocale) {
  …
}
```

For these Annotations to Work…

We need to 'enable them', just like for Java-based configuration

```
<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="...">

    <context:annotation-config/>

    <bean id="transferService" class="transfer.TransferService""/>
    <bean id="accountRepository" class="transfer.JdbcAccountRepository""/>

</beans>
```

Turns on checking of DI annotations for beans declared **in this context**

Topics in this Session

- Configuration Approaches
- Java-based Configuration
- Annotation Quick Start
- Other Configuration Annotations
- **Component Scanning**
- Stereotype and Meta-annotations
- When To Use What
- Summary & Lab
- Advanced Options

@Component

- Identifies POJOs as Spring Beans
- Removes the need to specify *almost anything* in XML

```
@Component
public class TransferServiceImpl implements TransferService
    @Autowired
    public TransferServiceImpl(AccountRepository ar) {
        this.accountRepository = ar;
    }
    …
}
```

@Component and names

- Default bean name is de-capitalized non-qualified name
- @Component takes an optional String to name the bean
  - Arguably not a best practice to put bean names in your Java code

```
@Component("transferService")  ")
public class TransferServiceImpl implements TransferService
    @Autowired
    public TransferServiceImpl(AccountRepository ar) {
        this.accountRepository = ar;
    }
    …
}
```

For This Annotation To Work…

- We need to enable ‘component scanning’
  - Auto-enables <context:annotation-config/>

```
<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-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:component-scan base-package="transfer""/>

</beans>
```

No more need to mention the beans in the application context

@Scope and @Primary

Like with @Bean:

- @Scope allows customizing the instantiation strategy

```
@Scope("prototype") @Component
public class TransferServiceImpl implements TransferService {
    ...
```

- @Primary allows making one component-scanned bean the primary candidate for autowiring

```
@Primary @Component
public class DefaultCalculationService implements CalculationService {
    ...
```

Topics in this Session

- Configuration Approaches
- Java-based Configuration
- Annotation Quick Start
- Other Configuration Annotations
- Component Scanning
- **Stereotype and Meta-annotations**
- When To Use What
- Summary & Lab
- Advanced Options

Stereotype annotations

- Component scanning also checks for annotations that are themselves annotated with @Component
  - Identify the architectural purpose of a class
  - Particularly useful for matching (e.g. with AOP)
- Spring framework stereotype annotations:
  - @Service – identify service classes
  - @Repository – identify data access classes
  - @Controller – identify web presentation classes
- Other Spring-based frameworks define their own

Meta-annotations

- A meta-annotation is an annotation which can be used to annotate other annotations
- e.g. all of your service beans should be configurable using component scanning and be transactional

```
@Service
@Transactional(timeout=60)
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface MyTransactionalService {
   String value() default "";
}
```

```
@MyTransactionalService
public class TransferServiceImpl implements TransferService {
...
```

Topics in this Session

- Configuration Approaches
- Java-based Configuration
- Annotation Quick Start
- Other Configuration Annotations
- Component Scanning
- Stereotype and Meta-annotations
- **When To Use What**
- Summary & Lab
- Advanced Options

When use what?

Annotations:

- Nice for frequently changing beans
  - Start using for small isolated parts of your application (e.g. Spring @MVC controllers)
- Pros:
  - Single place to edit (just the class)
  - Allows for very rapid development
- Cons:
  - Configuration spread across your code base
  - Only works for your own code

  --Maintenance issues

  --Recompilation for any changes

When use what?

XML:

- For infrastructure and more 'static' beans
- Pros:

  – Is centralized in one (or a few) places
  – Most familiar configuration format for most people
  – Can be used for all classes (not just your own)

- Cons:

  – Some people just don't like XML...
  -Changes in bean classes must be synchronized immediately to xml or else
  -XML could go out of sync with beans(java files)

When use what?

Java-based:

- For full control over instantiation and configuration
  - As alternative to implementing FactoryBean
- Pros:
  - Configuration still external to code
  - Integrates well with legacy code
    - Can call methods such as *addDataSource(DataSource d)*
- Cons:
  - Wiring won't show up in STS
  - No equivalent to the XML namespaces

Topics in this Session

- Configuration Approaches
- Java-based Configuration
- Annotation Quick Start
- Other Configuration Annotations
- Component Scanning
- Stereotype and Meta-annotations
- When To Use What
- **Summary & Lab**
- Advanced Options

Summary

- Spring's configuration directives can be written in XML, using Java or using annotations
- You can mix and match XML, Java and annotations as you please
- Autowiring with @Component or @Configuration allows for almost empty XML configuration files