Spring 3.0 Features

Cognizant

Passion for building stronger businesses

C3: Protected

About the Author

Created By:	Ramesh C.P(161646)
Credential Information:	SCJP, 8+ years of Experience in technical training
Version and Date:	SPRING/PPT/1011/3.0

Cognizant Certified Official Curriculum

UNDER THE SEAL OF COGNIZANT ACADEMY

Icons Used

Questions

Tools

Hands on Exercise

Coding Standards

Test Your Understanding

Reference

Demonstration

A Welcome Break

Contacts

Spring Overview Review

- Spring is an Open Source application framework
  - Facilitates development of Java EE applications
  - http://www.springframework.org
- Spring services can be used in Java SE and Java EE applications
- Created by Rod Johnson with Juergen Hoeller and others out of a frustration with EJBs and other J2EE technologies
- Spring is a light-weight, non-invasive framework
  - Applies to all architectural tiers of a Java EE application

Goals of Spring Framework

- Non-invasive
  - Application code is independent of the framework
  - Selectively apply Spring to existing application
  - Spring dependencies minimized
- Spring usable in any environment
  - With or without a Java EE container
  - With or without JNDI
  - With JDBC, JTA, Hibernate or other transaction models
- Facilitates good coding practices
  - Encourages coding with interfaces instead of classes

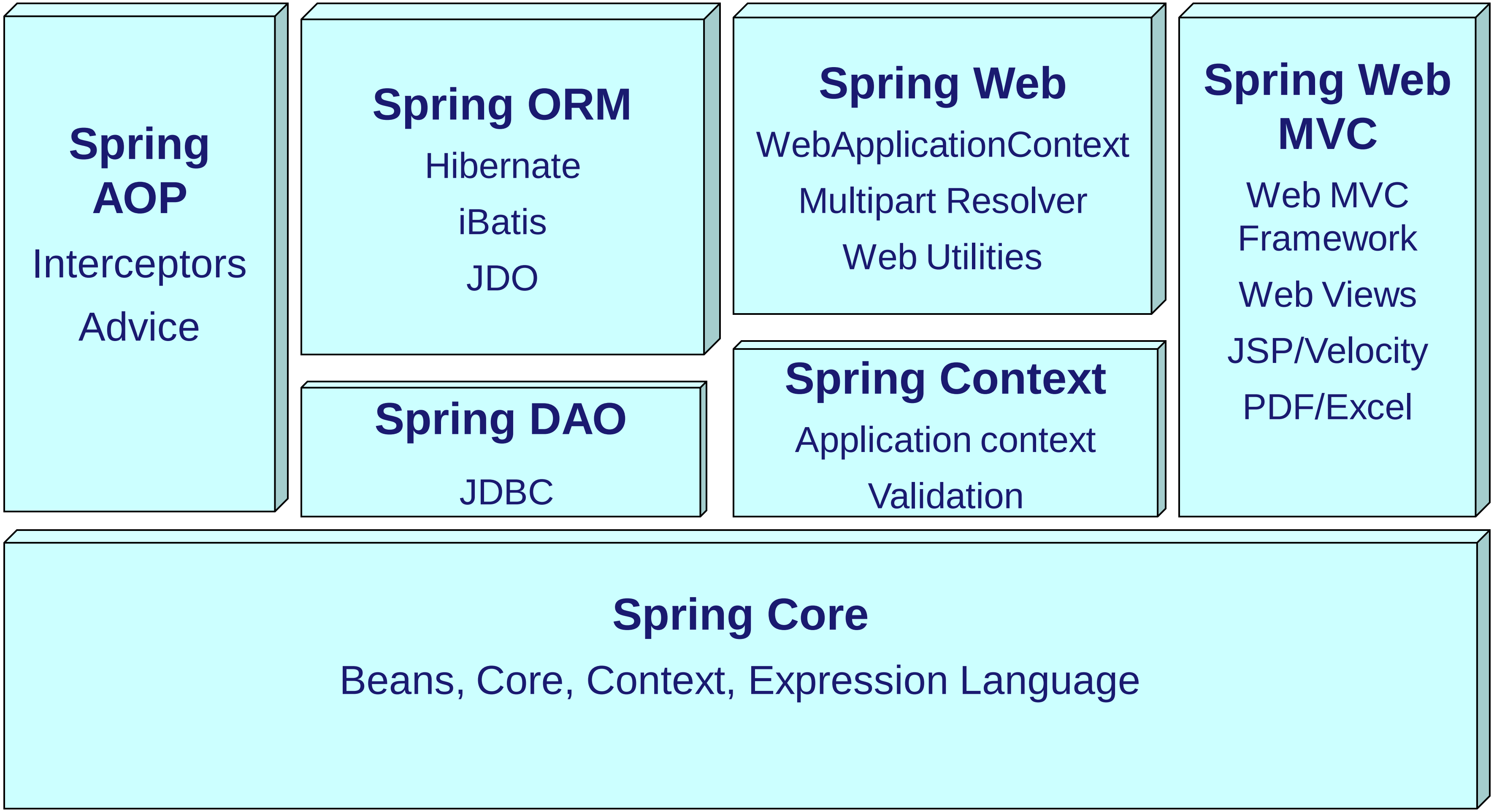
Goals of Spring Framework (Contd.)

- Promote plugability
  - Injected services can be replaced easily
  - XML or property files configure applications
  - Result - maintainable and reusable components within an "integration" framework
- Applications are easy to test
  - Most objects are POJOs
  - Mock objects supported through Dependency Injection
- Provide a consistent framework
  - Consistent approach is used across different parts of framework

Key Features of Spring (Contd.)

- Transaction abstraction
  - Consistent model for a variety of transaction services: JDBC, JTA, Hibernate, etc....
- MVC Framework for creating Web Apps
  - Uses of Dependency Injection (DI)
  - Can be used in place of or in combination with Struts and/or JSF
- Lightweight remoting (as opposed to EJB)
  - POJO-based remoting uses a variety of protocols
  - Provides a consistent, abstracted architectural approach to remote access

Spring Architecture



Spring AOP

Interceptors

Advice

Spring ORM

Hibernate

iBatis

JDO

Spring DAO

JDBC

Spring Web

WebApplicationContext

Multipart Resolver

Web Utilities

Spring Context

Application context

Validation

Spring Web MVC

Web MVC Framework

Web Views

JSP/Velocity

PDF/Excel

Spring Core

Beans, Core, Context, Expression Language

Key Features of Spring

- IoC container
  - Configuration management of POJOs
- Aspect-Oriented Programming (AOP) Framework
  - Services are modularized (out-of-the-box or custom)
  - Services are applied to the application in a declarative manner
  - Comprehensive coverage of all tiers
- Data access abstraction
  - Consistent architectural approach to data access
  - Independent of particular persistence products
  - Simplifies use of JDBC and other data access APIs

Configuring Spring

- Traditionally Spring was configured in external XML files
  - The files described the beans and their relationships to other beans
- Spring 3 provides alternatives to configuration
  - XML files
  - Annotations in Spring beans
  - Java Code configuring beans

POJOS and Interfaces – Problems

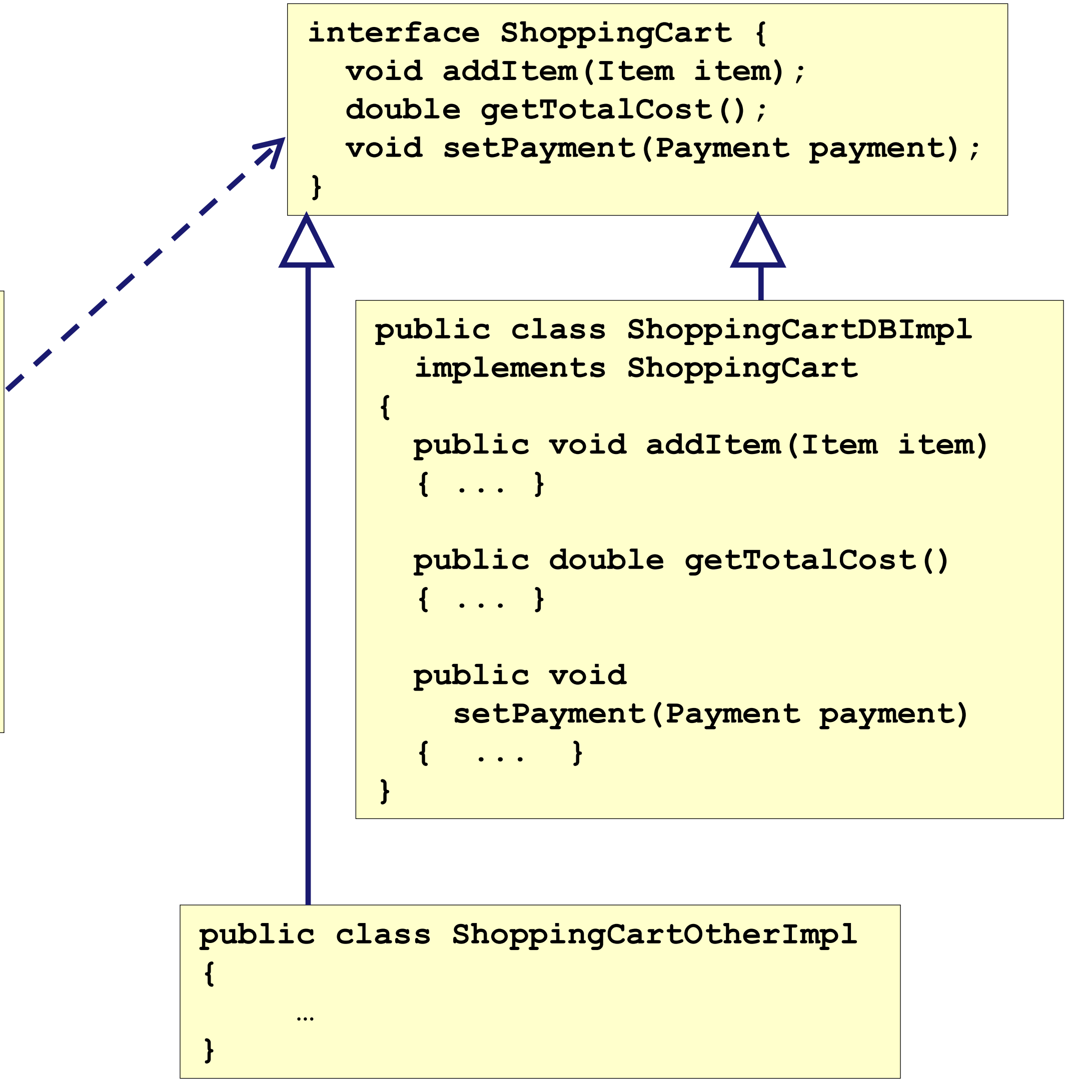
- The Client code can code to interfaces EXCEPT for creating the object
  - The Client must then know "which object to create"
- Some objects need a lot of initialization. For example, "which database"
  - Where do you put this info?
    - Choices are:
      - The Constructor
      - Some other code calling a "setter" on the object
      - Use JNDI
      - Properties file
      - The Spring way with Dependency Injection

Closer Look at POJOs and Interfaces

- Key parts of a POJO
  - An interface
  - Properties

```
// Client Code.
// Deal only with the
// ShoppingCart interface

// We'll fill this in later
ShoppingCart cart = ???;

cart.addItem(new Item("sku1234"));
double cost = cart.getTotalCost();
Payment payment = new Cash(cost);
cart.checkOut(payment);
```

- Who chooses which Shopping Cart to use?
- How is the DB init'd?



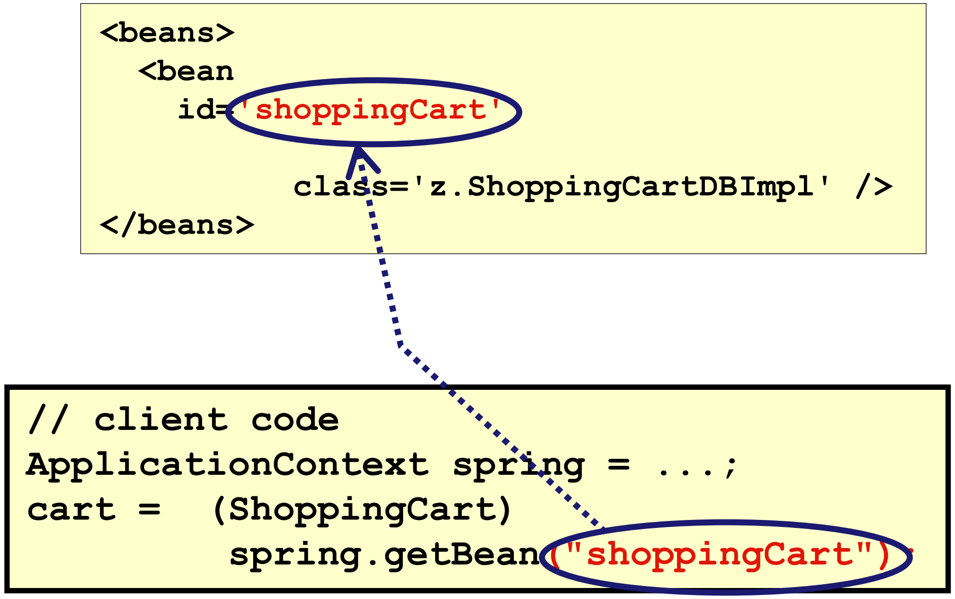
```
interface ShoppingCart {
  void addItem(Item item);
  double getTotalCost();
  void setPayment(Payment payment);
}
```

```
public class ShoppingCartDBImpl
   implements ShoppingCart
{
   public void addItem(Item item)
   { ... }

   public double getTotalCost()
   { ... }

   public void
      setPayment(Payment payment)
   {  ...  }
}
```

```
public class ShoppingCartOtherImpl
{
        …
}
```

Spring is an Object Factory(XML)

- Client code asks Spring to provide the correct object
- Spring looks up the correct object in an XML configuration file



```
<beans>
  <bean
    id='shoppingCart'

          class='z.ShoppingCartDBImpl' />
</beans>
```

```
// client code
ApplicationContext spring = ...;
cart =  (ShoppingCart)
          spring.getBean("shoppingCart");
```

```
package z;
public class ShoppingCartDBImpl
  implements ShoppingCart
{
  public void addItem(Item item)
  { ... }

  public void
    checkOut(Payment payment)
  {   ... }
}
```

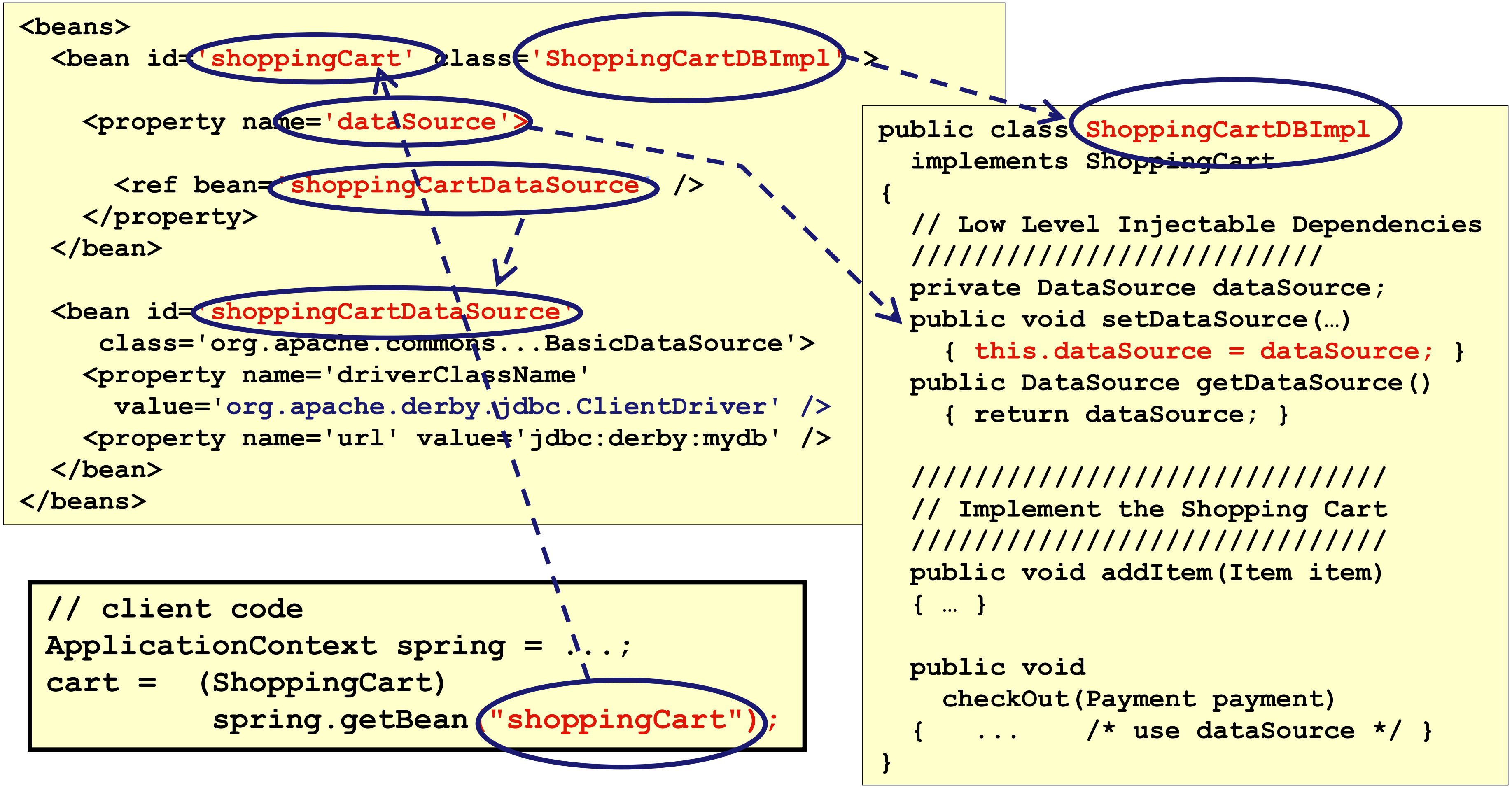
Dependency Injection

- 'Push' configuration - objects are given their companion components; they don't "look" for their companions
  - Instead of building or locating the objects code would need, the framework would push the objects into their locations
- Spring resolves dependencies
  - Between multiple objects
  - Between objects and configuration parameters

Dependency Injection - Example

- How to give ShoppingCartDBImpl its database?
- Solution:
  - Add dataSource property
    - setDataSource() and getDataSource() methods
  - Configure the value of the dataSource in the Spring config file
  - We will use an Apache Commons DataSource
    - org.apache.commons.dbcp.BasicDataSource
  - Configure DataSource in the Spring config file

DI Example (Contd.)



```
<beans>
  <bean id='shoppingCart' class='ShoppingCartDBImpl'>

    <property name='dataSource'>

      <ref bean='shoppingCartDataSource' />
    </property>
  </bean>

  <bean id='shoppingCartDataSource'
     class='org.apache.commons...BasicDataSource'>
    <property name='driverClassName'
      value='org.apache.derby.jdbc.ClientDriver' />
    <property name='url' value='jdbc:derby:mydb' />
  </bean>
</beans>
```

```
public class ShoppingCartDBImpl
  implements ShoppingCart
{
  // Low Level Injectable Dependencies
  //////////////////////////
  private DataSource dataSource;
  public void setDataSource(…)
    { this.dataSource = dataSource; }
  public DataSource getDataSource()
    { return dataSource; }

  //////////////////////////////
  // Implement the Shopping Cart
  //////////////////////////////
  public void addItem(Item item)
  { … }

  public void
    checkOut(Payment payment)
  {   ...    /* use dataSource */ }
}
```

```
// client code
ApplicationContext spring = ...;
cart =  (ShoppingCart)
          spring.getBean("shoppingCart");
```

DI Example (Contd.)

- Here's how this works:

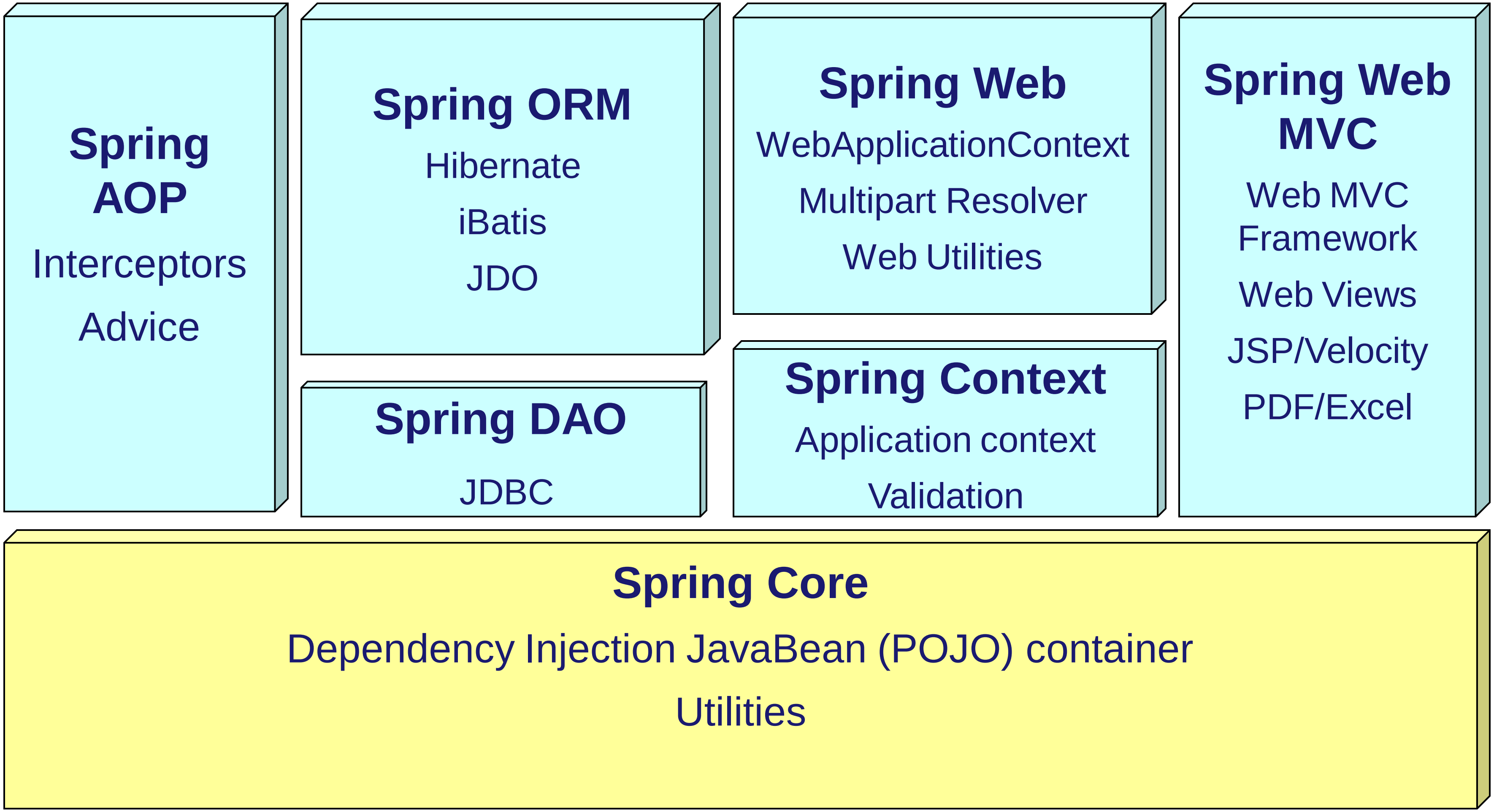
1. The Client asks Spring for a "shoppingCart".
2. Spring looks for a bean named "shoppingCart" in an XML config file.
3. Spring determines that it must instantiate a copy of ShoppingCartDBImpl.
4. Spring determines that it must initialize the property named dataSource with an object named shoppingCartDataSource.
5. Spring creates an instances of the apache commons BasicDataSource.
6. Spring initializes the driverClassName and url properties of BasicDataSource with the given String constants.
7. Spring sets the value of the DataSource property of the ShoppingCartDBImpl to be the configured apache DataSource.
8. Spring returns the configured ShoppingCartDBImpl to the client.

- Notice that the ShoppingCartDBImpl has no dependencies on Spring.
- The client has Spring dependencies. However, it does not have dependencies on the classes that implement the ShoppingCart interface.

Dependency Injection (Contd.)

- Dependency Injection also called "Inversion of Control"
  - The environment configures objects rather than objects configuring themselves in constructors
- Uses Java language constructs – setters and properties
- Two types of Dependency Injection
  - Constructor injection
  - Setter Injection
- Spring supports several creational patterns
  - Singleton
  - Prototypes
  - Custom

Spring Architecture



Spring AOP

Interceptors

Advice

Spring ORM

Hibernate

iBatis

JDO

Spring DAO

JDBC

Spring Web

WebApplicationContext

Multipart Resolver

Web Utilities

Spring Context

Application context

Validation

Spring Web MVC

Web MVC Framework

Web Views

JSP/Velocity

PDF/Excel

Spring Core

Dependency Injection JavaBean (POJO) container

Utilities

Spring Jars

- Spring 2.x provided a single Spring.jar which contained the majority of the Spring framework
  - Spring 3.0 provides a dist folder with a set of jars
  - A particular Spring project will require a unique set of Spring jars
- In addition to Spring jars every project will need a set of dependant jars
  - The dependent jars can come from a variety of sources
    - Apache
    - Oracle
    - Coucho
- Resolving dependencies within a Spring project is often very difficult
  - Build tools like Maven or Ivy can help resolve the dependencies

Spring DI Container

- Objects are created by Factories
- The (Java)Bean factory interface is
  - org.springframework.beans.factory.BeanFactory
- Usually, this more capable sub-interface is used:
  - org.springframework.context.ApplicationContext
- Two common implementation classes are:
  - org.springframework.context.support.ClassPathApplicationContext
  - org.springframework.context.support.FileSystemApplicationContext

Spring DI Container (Contd.)

- ApplicationContext is recommend over BeanFactory
  - More features
  - Automatically recognizes pre/post processors
  - MessageSource supports message localization
  - Supports application events and listeners
  - ResourceLoader used for handling low-level resources
- Spring configuration can be provided:
  - In XML format (most commonly used)
  - By property files
  - Programmatically

Initializing the Container

- ApplicationContext can be initialized in various ways
  - Pointing to a single resource on the classpath
  - ApplicationContext context =
  - new ClassPathXmlApplicationContext("Context.xml");

  - Pointing to a single resource on the file system
  - ApplicationContext context =
  - new FileSystemXmlApplicationContext("/SomeDir/Context.xml");

  - Referencing multiple resource fragments
  - String[] resources = {"Context.xml","OtherResource.xml"};
  - ApplicationContext context =
  - new ClassPathXmlApplicationContext(resources);

  - Loading all available resources on classpath
  - ApplicationContext context = new
  - ClassPathXmlApplicationContext("classpath*:Context.xml");

Accessing Beans from the Container

- Beans are obtained from the factory
  - Using the getBean(...) method
  - ReservationBO reservation = (ReservationBO)context.getBean("reservation");
  - Factory resolves bean dependencies before returning instance
  - getBean in Spring 3 has several different forms:
    - T getBean(Class<T> requiredType)
    - T getBean(String name, Class<T> requiredType)

Defining and Naming Beans

- Spring beans are declared using <bean> element
- Names given using either "id" or "name" attribute
- id attribute
  - <bean id="reservation" class="di.Reservation"/>
  - id value must be unique to the XML document
  - id must be a valid XML name
- name attribute
  - allows for one or more identifiers to be specified
  - Restrictions of id attribute do not apply
  - Multiple (comma separated) identifiers can be specified
  - <bean name="reservation,carRes" class="di.Reservation"/>
- Both id and name attributes
  - <bean id="r" name="reservation,carRes" class="di.Reserv"/>

Spring 3 Annotations

- ❖ Annotations are a alternative to configuration in XML
  - ◆ Added in Spring 2.5, enhanced in Spring 3
- ❖ Provides most of the functionality found in XML configuration
  - ◆ Some tasks are much easier in XML and some tasks are much easier in annotations
- ❖ Annotations are imported like classes
  - ◆ The annotation will precede whatever it is you are annotating

Spring 3.0 New Features Overview

- Ported to Java 1.5
  - Takes advantage of new Java features
- Spring Expression Language (SpEL)
- General purpose type conversion and formatting
- Declarative model validation
- RESTful web service supported (client & server)
- IoC Container enhancements
- Data tier enhancements
- Improved documentation
- Distribution format
- Build system

Java 1.5

- Spring 3.0 runs on and requires Java 1.5
- BeanFactory returns typed beans when possible

```
T getBean(Class<T> requiredType)
T getBean(String name, Class<T> requiredType)
Map<String,T> getBeansOfType(Class<T> type)
```

- TaskExecutor now extends **`java.util.concurrentExecutor`**
  - **`AsyncTaskExecutor`** supports Callables with Futures
- Type ApplicationListener<E>

Spring Expression Language

- Language for runtime object graph manipulation
- Works across all Spring products
- Designed to work with tools
  - E.g. code completion in Eclipse
- Similar in concept to OGNL, JBoss EL, etc.
- Parser / Evaluator API
- Support very dynamic interaction with Beans

```
org.springframework.expression
```

Spring Expression Language (2)

- Literals
- boolean and relational operators
- Regular expressions
- Access properties, arrays, lists and maps
- Method invocation
- Collection support
- Assignment
- Variables
- Templates
- Etc…

Spring Expression Language (3)

- ❖ Expression parser and evaluator

```
ExpressionParser parser = SpelExpressionParser();
Expression exp = parser.parseExpression(
                    “new String(‘hello world’).toUpperCase()”);
String message = exp.getValue(String.class);
```

Spring Expression Language (4)

- Use across Spring

```
<bean id=“numberGuess” class=“NumberGuess”>
  <property name=“randomNumber”
            value=“#{ T(java.lang.Math).random() * 100.0}”/>
  <!-- Other properties -->
</bean>

<bean id=“taxCalculator” class=“TaxCalculator”>
  <property name=“defaultLocale”
            value=“#{systemProperties[‘user.region’] }”/>
 <!-- Other properties -->
</bean>
```

General Purpose Type Conversion

- Alternate to ProperyEditors for IoC Container
- Used by SEL, IoC and DataBinder
- org.springframework.core.convert
- Used if ConversionSystem is registered
- Pattern:

```
Converter
ConverterFactory
ConversionService
```

Type Conversion (2)

- Converter Interface

```
public interface Converter<S,T> {
  T convert(S source);
}
```

Type Conversion (3)

- Converter Factory

```
public interface ConverterFactory<S,R> {
  <T extends R>Converter<S, T> getConverter(Class<T> targetType);
}
```

Field formatting system

- Alternative to PropertyEditors for Web tier
- Formatter, Printer and Parser Interfaces
- FormatterRegister, etc.
- Support annotation based format specification

Field Formatting System(2)

- Formatter Interface

```
public interface Formatter<T> extends Printer<T>, Parser<T> {
}
```

Field Formatting System(3)

- Printer Interface

```
public interface Printer<T> {
  String print(T fieldValue, Locale locale);
}
```

Field Formatting System(4)

- Parser Interface

```
public interface Parser<T> {
  T parse(String clientValue, Locale locale)
                                   throws ParseException;
}
```

Validation Enhancements

- JSR-303 Bean Validation supported
- DataBinder validation
- Spring MVC validation with @Controller inputs

JSR-303 Bean Validation

```
public class PersonForm {

   @NotNull
   @Size(max=64)
   private String name;

   @Min(0)
   private in age;
}
```

RESTful Web Services

- REST being adopted as scalable architectural style
- Spring 3.0 adds both Client and Server support for RESTful Web Services

RESTful Web Services – Client (2)

- RestTemplate
- HTTP Message Conversion

RESTful Web Services - Server (3)

- Comprehensive integrated support for REST
- URI Templates (Parsing URIs to arguments)
- Content Negotiation
- Views
  - AbstractAtomFeedView, AbstractRSSFeedView, MarshallingView
- HTTP Method Conversion
- ETag support (Shallow, not deep support)

IoC Container Enhancements

- Java based Bean metadata
  - Some features of JavaConfig added, e.g. @Configuration, @Bean, @Value
- Server

Data Tier

- Embedded database support
  - org.springframework.jdbc.datasource.embedded
    - Derby
    - H2
    - HSQL
- Object -> XML mapping now in data tier

  `org.spring.framework.oxm`

  - Marshall and Unmarshall Beans to XML
  - Supports variety of libraries
    - JAXB, Castor, XMLBeans, JiBX, XStream

Distribution Model

- One JAR file per Spring modules
  - Tailored deployments
  - spring.jar is no longer provided.
- Dependencies no longer provides
  - Must find and download each dependency
  - E.g., org.aopalliance-1.0.jar

New Build System

- Ported from Spring Web Flow 2.0
- Based on Ivy
- Consistent deployment procedure
- Consistent dependency management
- Consistent generation of OSGi manifests

Spring 3.0 Features

Time for a Break !

Spring 3.0 Features

- Questions from participants

Spring 3.0 Features: Summary

- Ported to Java 1.5
  - Takes advantage of new Java features
- Spring Expression Language (SpEL)
- General purpose type conversion and formatting
- Declarative model validation
- RESTful web service supported (client & server)
- IoC Container enhancements
- Data tier enhancements
- Improved documentation
- Distribution format
- Build system

Spring 3.0 Features: Source

- http://static.springsource.org/spring/docs/3.0.0.RELEASE/reference/html/spring-whats-new.html

Disclaimer: Parts of the content of this course is based on the materials available from the Web sites and books listed above. The materials that can be accessed from linked sites are not maintained by Cognizant Academy and we are not responsible for the contents thereof. All trademarks, service marks, and trade names in this course are the marks of the respective owner(s).

You have successfully completed Spring 3.0 Features

Click here to proceed

Cognizant

Passion for building stronger businesses