Introduction to the Spring Framework

Elements of the Spring Framework – everything you need

Professional programming – component based design

Inversion of Control – principles

Creating Components in Spring

Dependency Injection, Parameter Injection
Elements of the Spring Framework

**DAO**
Spring JDBC
Transaction Management

**ORM**
JPA
Hibernate
Toplink, iBatis

**JEE**
JMX
JMS
JCA
Remoting
EJBs
EMail

**WEB**
Spring MVC
Struts
Tapestry
JSF
Portlet
Velocity

**AOP**
Spring AOP - AspectJ Integration

**CORE**
The IoC Container

www.springframework.org
Component Based Design
- Agile Modeling Principles

**Acyclic Dependencies**
- Allow no cycles in the dependencies graph between components.
- For example disallow A -> B -> C -> A, because it includes a cycle.

**Common Closure**
- The classes of a component should be closed together against the same kinds of changes.
- A change that affects a class within a component should not affect classes outside that component.

**Common Reuse**
- The classes in a component are reused together.
- If you reuse one class in a component you reuse them all.

**Dependency Inversion**
- Abstractions should not depend on details, instead details should depend on abstractions.

**Open-Closed**
- Software elements should be open for extension but closed for modification.

www.agilemodeling.com
Component Based Programming
- General Principles

Reasons for Components
- Advantages: problem decomposition, testability, configurability, reusability
- Other related paradigms: interface based design, test driven development

Component integration
- How – component should know about the system, integration component
- When – before the first usage: not always easy to implement

Tipical design patterns: singleton, prototype, factory bean, static factory method, init method

Singleton – single instance of a component

```java
public class A {
    private static A a = null;
    public static A getInstance() {
        if (a == null) {
            a = new A();
        }
        return a;
    }
    ... // other methods
}
```
Component Based Programming - Design Patterns I.

Factory bean – knowledge of concrete components

interface A {
    ...
}

class D implements A {
    ....
}

public class F {
    public F(Config c) {
        // factory constructor
        // initialize F
    }
    public A getA(Param p) {
        A a = new D();
        // initialize ‘a’ according to ‘p’
        ... return a;
    }
}
Component Based Programming  
- Design Patterns II.

Static factory method – selection and configuration of the concrete component

```java
public class B {
    private B() {
        // initialize B
    }
    public static B getNewB(Param p) {
        B b = new B();
        // initialize ‘b’ according to ‘p’
        ....
        return b;
    }
}
```

Init method – initial configuration of the component

```java
public class C {
    void init() {
        // initialize C
    }
    // other methods
}
```
public static void main(String[] args) throws IOException {

    // open up standard input
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

    String name = null;
    String password = null;

    // prompt the user for name and password
    System.out.print("Enter your login name: ");
    name = br.readLine();
    System.out.print("Enter the password: ");
    password = br.readLine();

    // check if the login information is correct
    if (checkAccount(name, password)) {
        System.out.println("Welcome to the real world!");
    } else {
        System.out.println("Your login failed ... ");
    }

}
public class LoginServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {

        // get user name and password from the HTTP request's parameter
        String name = req.getParameter("NAME");
        String password = req.getParameter("PASSWORD");

        // check if the login information is correct
        if (checkAccount(name, password)) {
            req.getSession().putValue("USER", name);  // save the application state
            // go to the application page
            ...
        } else {
            // show the login page again
            ...
        }
    }
}
Inversion of Control - Fundamental Principles

Classical approach
- Stand-alone application using libraries
- The application determines the sequence of the events
- Components must use standard APIs
- Components must initialize the environment
- Components call library APIs to request services

Container framework
- Application components are running inside the container
- External events trigger application functions through the container
- Components must comply with certain interface specification
- The container instantiates the components with the environment set up
- The container provides services (lifecycle, communication, security, etc.)
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
  <bean id="bean1" class="...">
    <!-- collaborators and configuration for bean1 go here -->
  </bean>
  <bean id="bean2" class="...">
    <!-- collaborators and configuration for bean2 go here -->
  </bean>
  <!-- more bean definitions go here... -->
</beans>
Creating Components

ApplicationContext context = new ClassPathXmlApplicationContext(new String[]{
    "applicationContext.xml"});
ExampleBean bean = context.getBean("exampleBean");

Constructor:
<bean id="exampleBean" class="examples.ExampleBean"/>

Static factory method:
<bean id="exampleBean"
    class="examples.StaticFactoryMethodExampleBean"
factory-method="createMyInstance"/>

Factory bean:
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="sampleFactoryBean" class="FactoryExampleBean">
...
</bean>
<!-- the bean to be created via the factory bean -->
<bean id="exampleBean"
    factory-bean="sampleFactoryBean"
factory-method="createOtherInstance"/>
Dependency and Parameter Injection
- Constructor Injection

```xml
<bean id="exampleBean" class="examples.ExampleBean">
  <!-- constructor injection using the nested <ref/> element -->
  <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>
  <!-- constructor injection using the neater 'ref' attribute -->
  <constructor-arg ref="yetAnotherBean"/>
  <constructor-arg type="int" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

public class ExampleBean {
  private AnotherBean beanOne;
  private YetAnotherBean beanTwo;
  private int i;

  public ExampleBean(AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
    this.beanOne = anotherBean;
    this.beanTwo = yetAnotherBean;
    this.i = i;
  }
}
```
Dependency and Parameter Injection - Property Injection

```xml
<bean id="exampleBean" class="examples.ExampleBean">
  <!-- setter injection using the nested <ref/> element -->
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>
  <!-- setter injection using the neater 'ref' attribute -->
  <property name="beanTwo" ref="yetAnotherBean"/>
  <property name="integerProperty" value="1"/>
</bean>
<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>

public class ExampleBean {
  private AnotherBean beanOne;
  private YetAnotherBean beanTwo;
  private int i;

  public void setBeanOne(AnotherBean beanOne) { this.beanOne = beanOne; }

  public void setBeanTwo(YetAnotherBean beanTwo) { this.beanTwo = beanTwo; }

  public void setIntegerProperty(int i) { this.i = i; }
}
```
Collection Properties

```xml
<bean id="moreComplexObject" class="example.ComplexObject">
  <!-- results in a setAdminEmails(java.util.Properties) call -->
  <property name="adminEmails">
    <props>
      <prop key="administrator">administrator@somecompany.org</prop>
      <prop key="support">support@somecompany.org</prop>
      <prop key="development">development@somecompany.org</prop>
    </props>
  </property>
  <!-- results in a setSomeList(java.util.List) call -->
  <property name="someList">
    <list>
      <value>a list element followed by a reference</value>
      <ref bean="myDataSource"/>
    </list>
  </property>
  <!-- results in a setSomeSet(java.util.Set) call -->
  <property name="someSet">
    <set>
      <value>just some string</value>
      <ref bean="myDataSource"/>
    </set>
  </property>
</bean>
```
Collection Properties – Map

<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
  <map>
    <entry>
      <key>
        <value>yup an entry</value>
      </key>
      <value>just some string</value>
    </entry>
    <entry>
      <key>
        <value>yup a ref</value>
      </key>
      <ref bean="myDataSource" />
    </entry>
    <entry>
      <key>
        <value>yup a null value</value>
      </key>
      <value><null/></value>
    </entry>
  </map>
</property>
</bean>
Bean Initialization and Destruction

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

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

Destruction:
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>

public class ExampleBean {
    public void cleanup() {
        // do some destruction work (like releasing pooled connections)
        ...
    }
}
Bean Scopes

General Application Contexts
- Singleton Scopes (scope="singleton")
  a single bean definition to a single object instance per Spring IoC container

- Prototype Scopes (scope="prototype")
  a single bean definition to any number of object instances

Web-aware Spring Application Contexts
- Request Scopes (scope="request")
  a single bean definition to the lifecycle of a single HTTP request; i.e. each and every HTTP request will have its own instance

- Session Scopes (scope="session")
  a single bean definition to the lifecycle of a HTTP Session

- Global Session Scopes (scope="globalSession")
  a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context
Bean, BeanFactory, ApplicationContext

Bean definition:
- Main components in the application, managed by the Spring IoC container.
- Has a name, which is a unique id in the application.
- Created via constructor, static factory method or factory bean.

The BeanFactory features:
- Central registry of application components.
- Centralizes configuration of application components.
- Holds bean definitions, each uniquely identified by a String name.
- Independent instance of a contained object (the Prototype design pattern).
- Single shared instance (the Singleton design pattern).

An ApplicationContext provides:
- Bean factory methods for accessing application components. Inherited from BeanFactory.
- The ability to load file resources in a generic fashion. Inherited from ResourceLoader interface.
- The ability to publish events to registered listeners. Inherited from ApplicationEventPublisher interface.
- The ability to resolve messages (internationalization). Inherited from MessageSource interface.
- Inheritance from a parent context. Definitions in a descendant context will always take priority
- Single parent context can be used by an entire web application.
- Each servlet has its own child context that is independent of that of any other servlet.
Sample Application DupeCheck Classes

```
class matching

spring::DupeCheck

- nameMatcher: NameMatcher
- segmentMatcher: SegmentMatcher

+ DupeCheck()
+ checkDupe(Reservation, Reservation) : boolean
+ getNameMatcher() : NameMatcher
+ setNameMatcher(NameMatcher) : void
+ getSegmentMatcher() : SegmentMatcher
+ setSegmentMatcher(SegmentMatcher) : void

spring::Main

+ Main()
+ main(String[]) : void

«interface»

SegmentMatcher

+ similar(Segment, Segment) : boolean

«interface»

NameMatcher

+ same(Passenger, Passenger) : boolean

FullNameMatcher

+ fullNameMatcher()
+ same(Passenger, Passenger) : boolean

SimpleSegmentMatcher

- hourDiff: int

+ SimpleSegmentMatcher()
+ similar(Segment, Segment) : boolean
+ getHourDiff() : int
+ setHourDiff(int) : void
```
Sample Application Data Model

**Passenger**
- title: Title
- lastName: String
- firstName: String

+ Passenger()
+ getTitle(): Title
+ setTitle(Title): void
+ getLastName(): String
+ setLastName(String): void
+ getFirstName(): String
+ setFirstName(String): void
+ toString(): String

**Reservation**
- airline: String
- passenger: Passenger
- segments: Set<Segment>

+ Reservation()
+ getAirline(): String
+ setAirline(String): void
+ getPassenger(): Passenger
+ setPassenger(Passenger): void
+ getSegments(): Set<Segment>
+ setSegments(Set<Segment>): void
+ toString(): String

**Segment**
- departure: Date
- flight: String
- bookingClass: BookingClass

+ Segment()
+ getDeparture(): Date
+ setDeparture(Date): void
+ getFlight(): String
+ setFlight(String): void
+ getBookingClass(): BookingClass
+ setBookingClass(BookingClass): void
+ toString(): String

**Passenger::Title**
- enum
  - MR
  - MRS
  - MS
  - DR

**Segment::BookingClass**
- enum
  - F
  - B
  - E