JAVA DAVE ENVIRONMENT

Creating web service, SOAP, P/L SQL & More…

Java Boot Camp, August, 2017.

Dr. Kishore Biswas (Forrest/柯修)

PhD. Artificial Intelligence—Natural Language Processing.

CanWay IT Training ®.

CONTENTS

1) Creating Web services (SOAP)
2) Spring framework review…
3) Code example review

1) Creating Web Services

EXAMPLE PRACTICE

- Create a soap web service and consume that service.

2) Spring Framework Review

1.1) Spring Framework

- Spring is a non-invasive and portable framework that allows us to introduce as much or as little as you want to your application.
- Promotes decoupling and reusability
- POJO Based
- Allows developers to focus more on reused business logic and less on plumbing problems.
- Reduces or alleviates code littering, ad hoc singletons, factories, service locators and multiple configuration files.
- Removes common code issues like leaking connections and more.
- Built in aspects such as transaction management
- Most business objects in Spring apps do not depend on the Spring framework.

1.1.1) Spring Framework Goals...

- Make J2EE easier to use
- Address end-to-end requirements rather than one tier
- Eliminate need for middle tier “glue”
- Provide the best *Inversion of Control* solution
- Provide a pure Java *AOP* implementation, focused on solving common problems in J2EE
- Fully portable across application servers
  - Core container can run in *any* environment, not only a server
  - Works well in WebSphere
- OO design is more important than any implementation technology, such as J2EE.
- Testability is essential, and a framework such as Spring should help make your code easier to test.
- Spring should not compete with good existing solutions, but should foster integration.

1.1.1) Spring Framework Goals…

- "**Non-invasive" framework**
  - Application code has minimal or *no* dependency on Spring APIs
  - **Key principal seen throughout Spring's design**
  - More power to the POJO
- *Facilitate unit testing*
  - Allow effective TDD
  - Allow business objects to be unit tested outside the container
- *Facilitate OO best practice*
  - We're used to implementing "EJB" or "J2EE" applications rather than OO applications.
  - **It doesn't have to be this way.**
- Provide a good alternative to EJB for many applications
- *Enhance productivity compared to "traditional" J2EE approaches*

1.1.1) Spring Framework Goals...

- Spring *complements* application servers like WebSphere
  - Spring comes out of the *application* space, rather than the *server* space
- Spring avoids the need for costly in-house frameworks
  - Can produce real savings
  - Focus your developers on *your* domain
  - Well-understood, generic, high-quality solution
  - Facilitates testability, increase productivity
- Provides a simpler, yet powerful, alternative to the EJB component model in many applications
  - But also plays well with EJB if you prefer to stick with EJB

1.2) Why Spring? ...

- Spring's overall theme: Simplifying Enterprise Java Development
- Strategies
  - Loose coupling with dependency injection
  - Declarative programming with AOP
  - Boilerplate reduction with templates
  - Minimally invasive and POJO-oriented

1.2) Why Spring?...

- Introduced to Spring by way of Hibernate
- Originally wanted Spring to provide a way to simplify DAO objects and provide declarative transaction support to our non-EJB applications.
- Needed a solution to loosely couple business logic in a POJO fashion.
- Wanted to build portable applications that provided clearer separation of presentation, business, and persistence logic.
- Easily integrated with our existing frameworks
- Great documentation and community support

1.2) Why Spring? : Simplifying Code With Spring

- Enables you to stop polluting code
- No more custom singleton objects
  - Beans are defined in a centralized configuration file
- No more custom factory object to build and/or locate other objects
- DAO simplification
  - Consistent CRUD
  - Data access templates
  - No more copy-paste try/catch/finally blocks
  - No more passing Connection objects between methods
  - No more leaked connections
- POJO Based
- Refactoring experience with Spring
- Caution Spring is addictive!

1.3.1) Dependency Injection Defined

- Method to create needed dependencies or look them up somehow without doing it in the dependent code
  - Often called Inversion of Control (IoC)
- IoC injects needed dependencies into the object instead
  - Setters or Constructor
- Primary goal is reduction of dependencies in code
  - an excellent goal in any case
  - This is the central part of Spring

1.3.2) Aspect Oriented Programming Defined

- Attempts to separate concerns, increase modularity, and decrease redundancy
  - Separation of Concerns (SoC)
    - Break up features to minimize overlap
  - Don’t Repeat Yourself (DRY)
    - Minimize code duplication
  - Cross-Cutting Concerns
    - Program aspects that affect many others (e.g. logging)
- AspectJ is the top AOP package
  - Java like syntax, IDE integration

1.3.3) Spring IOC + AOP

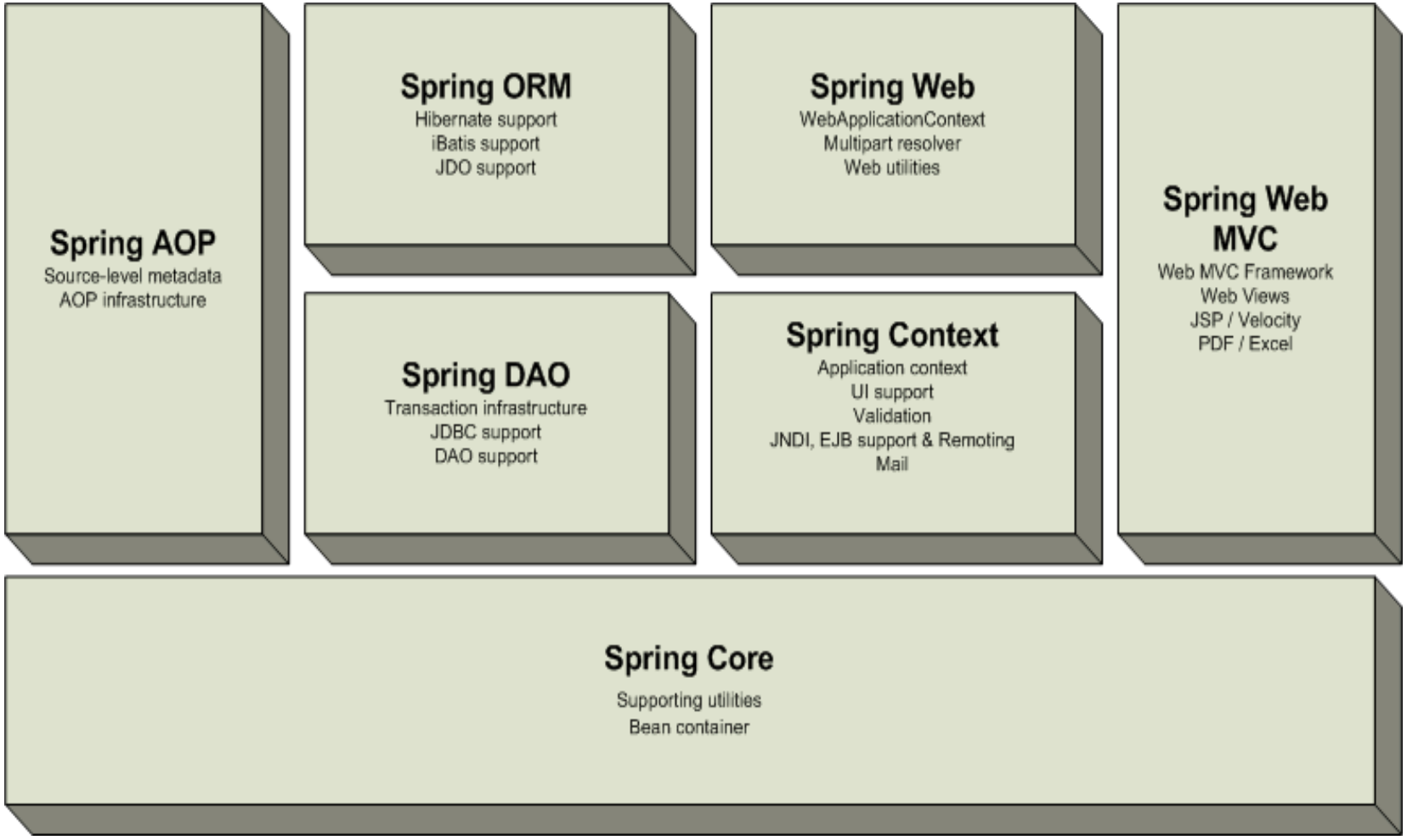
- IoC container
  - Setter based and constructor based dependency injection
  - Portable across application servers
  - Promotes good use of OO practices such as programming to interfaces.
  - Beans managed by an IoC container are reusable and decoupled from business logic
- AOP
  - Spring uses Dynamic AOP Proxy objects to provide cross-cutting services
  - Reusable components
  - Aopalliance support today
  - Integrates with the IoC container
  - AspectJ support in Spring 1.1

1.3.4) Portable Service Abstractions Defined

- Services that easily move between systems without heavy reworking
  - Ideally easy to run on any system
  - Abstraction without exposing service dependencies
    - LDAP access without knowing what LDAP is
    - Database access without typical JDBC hoops
- Basically everything in Spring that is not IoC or AOP

1.3.5) Spring Stack

Spring code structure



1.3.6) A Layered Framework

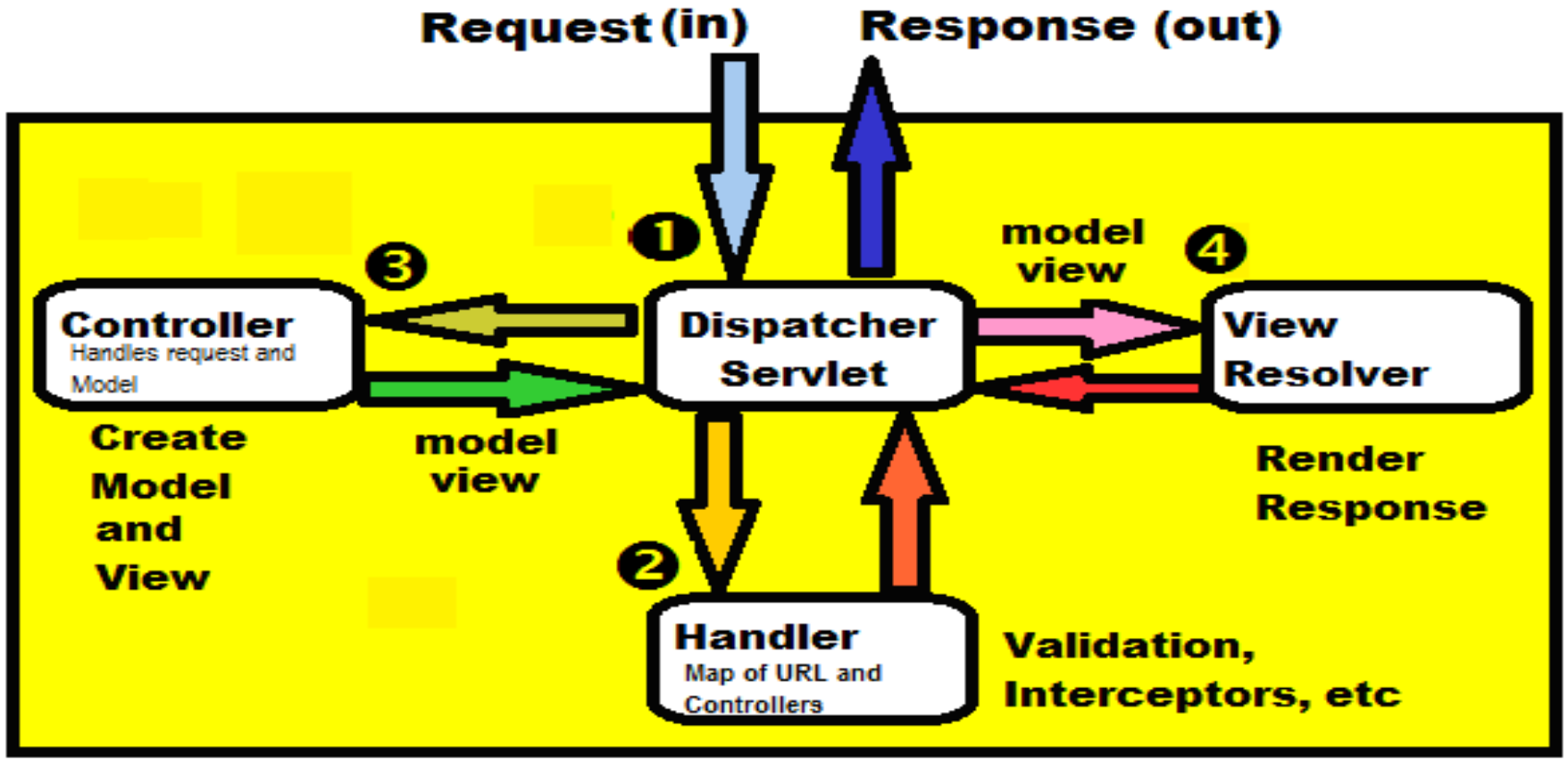
- Web MVC
- AOP framework
  - Integrates with IoC
- **IoC container**
  - **Dependency Injection**
- Transaction management
- Data access
- *One stop shop but can also use as modules*

1.3.7) Web MVC

- Most similar in design to Struts
  - Single shared Controller instance handles a particular request type
- *Controllers, interceptors run in the IoC container*
  - Important distinguishing feature
  - Spring eats its own dog food

1.3.8) Spring MVC

- Spring MVC operates via 4 key steps…



2) Difference between Spring Framework and JEE.

2.1) "Old" J2EE vs Spring...

- Write a SLSB
  - Home interface
  - Component interface
  - "Business methods" interface
  - Bean implementation class
  - Complex XML configuration
  - POJO delegate behind it if you want to test outside the container
  - Much of this is working around EJB
  - *If you want parameterization it gets even more complex*
    - *Need custom code, IoC container or "environment variables" !*

- Implement a Spring object
  - Business interface
  - Implementation class
  - Straightforward XML configuration
  - The first two steps are necessary in Java anyway
  - *If you want to manage simple properties or object dependencies, it's easy*

2.1) "Old" J2EE vs Spring

- Use your SLSB
  - Write a Service Locator and/or Business Delegate: need JNDI code
  - Each class that uses the service needs to depend on EJB interface (home.create) *or* you need a Business Delegate with substantial code duplication
  - *Hard to test outside a container*

- Use your Spring object
  - Just write the class that uses it in plain old Java
  - Express a dependency of the business interface type using *Java* (setter or constructor)
  - Simple, intuitive XML configuration
  - *No lookup code*
  - *Easily test with mock object*

2.2) Spring J2EE

- No more JNDI lookups
  - `JndiObjectFactoryBean`
  - Generic proxy for DataSources etc.
- No more EJB API dependencies, even in code calling EJBs
  - EJB proxies
  - No more `home.create()`
  - No more Service Locators or Business Delegates
  - Codeless EJB access
    - Callers depend on Business Methods interface, not EJB API
    - Exposed via Dependency Injection (naturally)
- Maximize code reuse by minimizing J2EE API dependencies
- **Spring does *not* prevent you using the full power of J2EE**
  - **Full power of WebSphere lies under the covers**
  - **Spring makes it easier to use effectively**

2.3) Spring OOP

- No more Singletons
  - An antipattern as commonly used
- Program to interfaces, not classes
- Facilitates use of the Strategy pattern
  - Makes good OO practice much easier to achieve
- IoC Dependency Injection keeps the container from messing up your object model
  - Base object granularity on OO concerns, not Spring concerns
- Combine with transparent persistence to achieve a **true domain model**

2.4) Spring Productivity

- Less code to develop in house
- Focus on your domain
- Reduced testing effort
- Try it and you won't look back
- "Old" J2EE has a poor productivity record
  - Need to simplify the programming model, not rely on tools to hide the complexity

2.5) Productivity Dividend

- Spring removes unnecessary code
- You end with *no* Java plumbing code and relatively simple XML
  - If your Spring XML is complex, you're probably doing things you couldn't do the old way without extensive custom coding
- The old way you have lots of Java plumbing code *and* lots of XML
  - A lot of the XML is *not* standard
- Combine with Hibernate or JDO for transparent persistence and the advantage is huge

3) Spring IOC container.

3.1) Inversion of Control (IoC)...

- A framework like Spring is responsible for instantiating the objects and pass them to application code
  - A.K.A. IoC container, bean container
- Inversion of Control (IoC)
  - The application code is no longer responsible for instantiate an interface with a specific implementation
  - A.K.A. Dependency Injection

3.1) Inversion of Control (IOC)...

- Dependency injection
  - Beans define their dependencies through constructor arguments or properties
  - The container provides the injection at runtime
- “Don’t talk to strangers”
- Also known as the Hollywood principle – “don’t call me I will call you”
- Decouples object creators and locators from application logic
- Easy to maintain and reuse
- Testing is easier

3.1) Inversion of Control (IoC)

IoC is all about Object dependencies.

Traditional "Pull" approach:

- Direct instantiation
- Asking a Factory for an implementation
- Looking up a service via JNDI

"Push" approach:
Something outside of the Object "pushes" its dependencies into it. The Object has no knowledge of how it gets its dependencies, it just assumes they are there. The "Push" approach is called "Dependency Injection".

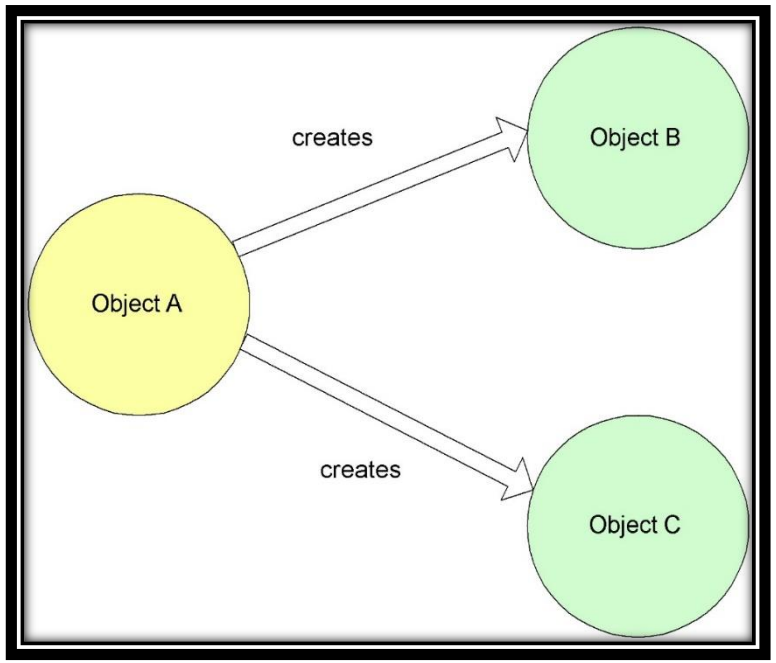
3.1.1) Pull Example

```
public class BookDemoServicePullImpl implements BookDemoService {


    public void addPublisherToBook(Book book) {
        BookDemoFactory factory = BookDemoFactory.getFactory();
        BookDemoDao dao = factory.getBookDemoDao();

        String isbn = book.getIsbn();
        if (book.getPublisher() == null && isbn != null) {
            Publisher publisher = dao.findPublisherByIsbn(isbn);
            book.setPublisher(publisher);
        }
    }
}
```

3.1.2) Push Example (Dependency Injection)

```
public class BookDemoServiceImpl implements BookDemoService {

    private BookDemoDao dao;

    public void addPublisherToBook(Book book) {
        String isbn = book.getIsbn();
        if (book.getPublisher() == null && isbn != null) {
            Publisher publisher = dao.findPublisherByIsbn(isbn);
            book.setPublisher(publisher);
        }
    }
    public void setBookDemoDao(BookDemoDao dao) {
        this.dao = dao;
    }
}
```

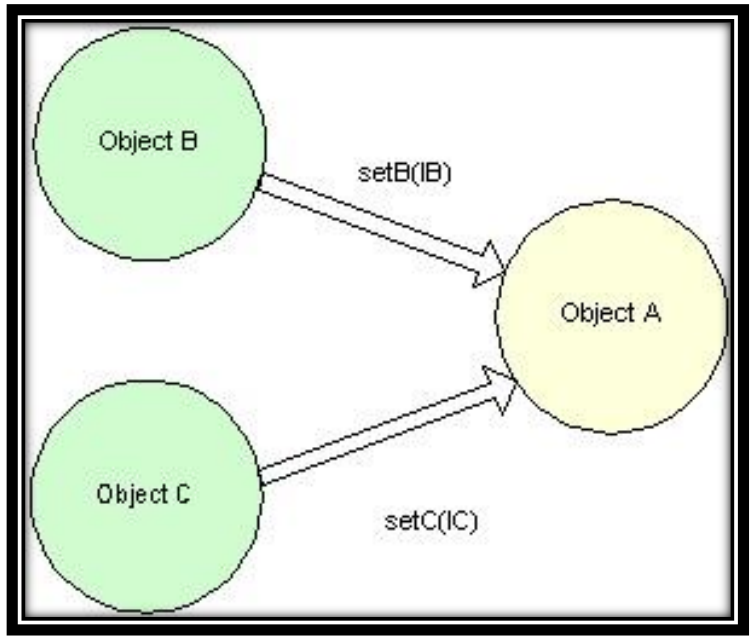
3.1.3) Non-IoC / Dependency Injection



(Non-IoC Service Object)

```
public class OrderServiceImpl implements
  IOrderService {
public Order saveOrder(Order order) throws
  OrderException{
  try{
  // 1. Create a Session/Connection object
  // 2. Start a transaction
  // 3. Lookup and invoke one of the methods in a
   // DAO and pass the Session/Connection object.
  // 4. Commit transaction
  }catch(Exception e){
  // handle e, rollback transaction, //cleanup, //
  throw e
  }finally{
  //Release resources and handle more exceptions
  }
}
```

3.1.4) IoC / Dependency Injection



(IoC Service Object)

```
public class OrderSpringService
  implements IOrderService {
IOrderDAO orderDAO;
public Order saveOrder(Order order)
  throws OrderException{
  // perform some business logic…
  return orderDAO.saveNewOrder(order);
}

public void setOrderDAO(IOrderDAO
  orderDAO) {
  this.orderDAO = orderDAO;
}
```

- Program to interfaces for your bean dependencies!

3.2) Why is Dependency Injection better?

2 reasons:

- Loose Coupling
- Testability

Loose Coupling is improved because you don't hard-code dependencies between layers and modules. Instead you configure them outside of the code. This makes it easy to swap in a new implementation of a service, or break off a module and reuse it elsewhere.

Testability is improved because your Objects don't know or care what environment they're in as long as someone injects their dependencies. Hence you can deploy Objects into a test environment and inject Mock Objects for their dependencies with ease.

3.3) How Spring does Inversion of Control

Write a configuration file in which you name concrete "beans" for the interfaces between your layers.

"Wire" the application together by stating which beans are dependent on each other.

Instantiate a Spring object called an ApplicationContext. This is a type of bean factory that will instantiate all your other beans and handle dependency injection.

3.4) Spring in the Middle Tier...

- Complete solution for managing business objects
  - Write your business objects as POJOs
  - Spring handles wiring and lookup
  - Simple, consistent, XML format (commonest choice)
  - But the IoC container is *not* tied to XML
- Application code has few dependencies on the container—often *no* dependencies on the container
  - Spring Pet Store has **no** dependencies on Spring IoC
  - No magic annotations for IoC: *nothing* Spring-specific
- Easy unit testing. TDD works!

3.4) Spring in the Middle Tier...

- **The most complete IoC container**
  - *Setter Dependency Injection*
    - Configuration via JavaBean properties
  - *Constructor Dependency Injection*
    - Configuration via constructor arguments
    - Pioneered by PicoContainer
  - *Dependency Lookup*
    - Avalon/EJB-style callbacks
  - I favour Setter Injection, but the Spring philosophy is that *you* make the choice
    - We are not ideological.
    - A good IoC container must provide sophisticated support for both injection models to allow use of legacy code

3.4.1) Middle Tier: Setter Injection

```
public class ServiceImpl implements Service {
   private int timeout;
   private AccountDao accountDao;

    public void setTimeout(int timeout) {
        this.timeout = timeout;
    }

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    // Business methods from Service
…

<bean id="service" class="com.mycompany.service.ServiceImpl">
   <property name="timeout"><value>30</value></property>
   <property name="accountDao"><ref local="accountDao"/></property>
</bean>
```

3.4.2) Middle Tier: Constructor Injection

```
public class ServiceImpl implements Service {
  private int timeout;
  private AccountDao accountDao;

   public ServiceImpl (int timeout, AccountDao accountDao) {
       this.timeout = timeout;
       this.accountDao = accountDao;
   }

   // Business methods from Service
```

```
<bean id="service" class="com.mycompany.service.ServiceImpl">
  <constructor-arg><value>30</value></constructor-arg>
  <constructor-arg><ref local="accountDao"/></constructor-arg>
</bean>
```

3.4.3) Middle Tier: Dependency Injection

- Simple or object properties
  - Configuration (timeout)
  - Dependencies on collaborators (accountDao)
- Configuration properties are also important
- Can run many existing classes unchanged
- “Autowiring”
- Trivial to test application classes outside the container, without Spring
- Can *reuse* application classes outside the container
- Hot swapping, instance pooling (with AOP)

3.5) Aspect Oriented Programming (AOP)...

- Complements OO programming
- Core business concerns vs. Crosscutting enterprise concerns
- Components of AOP
  - **Aspect** – unit of modularity for crosscutting concerns
  - **Join point** – well-defined points in the program flow
  - **Pointcut** – join point queries where advice executes
  - **Advice** – the block of code that runs based on the pointcut definition
  - **Weaving** – can be done at runtime or compile time. Inserts the advice (crosscutting concerns) into the code (core concerns).
- Aspects can be used as an alternative to existing technologies such as EJB. Ex: declarative transaction management, declarative security, profiling, logging, etc.
- Aspects can be added or removed as needed without changing your code.

3.5.1) Spring AOP...

- Framework that builds on the aopalliance interfaces.
- Aspects are coded with pure Java code. You do not need to learn pointcut query languages that are available in other AOP implementations.
- Spring aspects can be configured using its own IoC container.
  - Objects obtained from the IoC container can be transparently advised based on configuration
- Spring AOP has built in aspects such as providing transaction management, performance monitoring and more for your beans
- Spring AOP is not as robust as some other implementations such as AspectJ.
  - However, it does support common aspect uses to solve common problems in enterprise applications

3.5.1) Spring AOP...

- Supports the following advices:
  - method before
  - method after returning
  - throws advice
  - around advice (uses AOPAlliance MethodInterceptor directly)
- Spring allows you to chain together interceptors and advice with precedence.
- Aspects are weaved together at runtime. AspectJ uses compile time weaving.
- Spring AOP also includes advisors that contain advice and pointcut filtering.
- ProxyFactoryBean – sources AOP proxies from a Spring BeanFactory
- IoC + AOP is a great combination that is non-invasive

3.5.2) Spring AOP Around Advice Example...

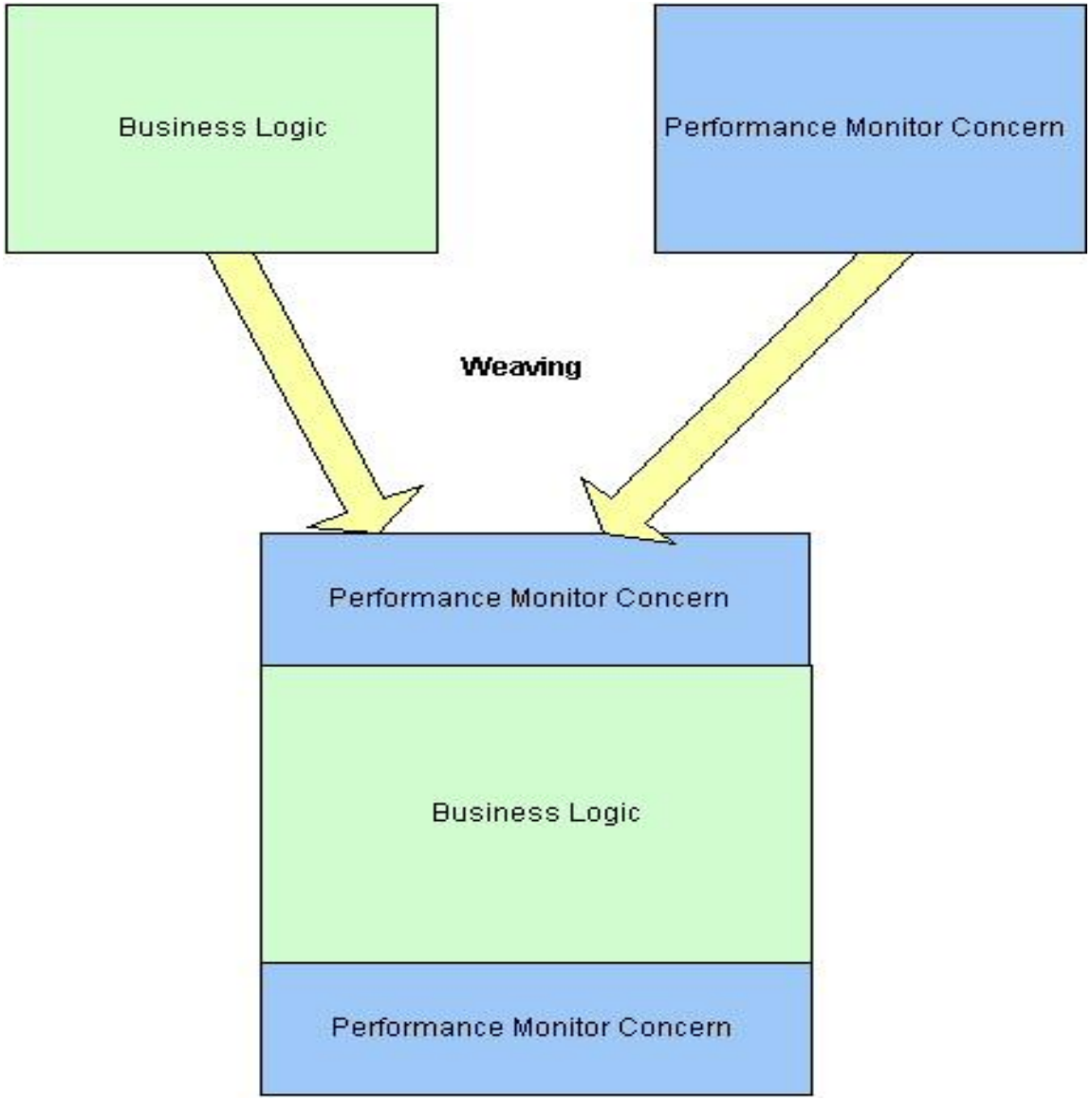
```
public class PerformanceMonitorDetailInterceptor implements MethodInterceptor {

    protected final Log logger = LogFactory.getLog(getClass());
    public Object invoke(MethodInvocation invocation) throws Throwable {
        String name =
                invocation.getMethod().getDeclaringClass().getName()
                        + "."
                        + invocation.getMethod().getName();

        StopWatch sw = new StopWatch(name);
        sw.start(name);

        Object rval = invocation.proceed();

        sw.stop();
        logger.info(sw.prettyPrint());
        return rval;
    }
```

3.5.2) Spring AOP Around Advice Example

- Advisor references the advice and the pointcut

```
<bean id="perfMonInterceptor"
   class=
   "com.meagle.service.interceptor.PerformanceMonitorDetailInterceptor"/>

<bean id="performanceAdvisor"
   class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

   <property name="advice">
          <ref local="perfMonInterceptor"/>
      </property>

      <property name="patterns">
          <list>
              <value>.*find.*</value>
              <value>.*save.*</value>
              <value>.*update.*</value>
          </list>
      </property>
</bean>
```

3.6.1) AOP Weaving



3.6.2) Spring AOP Key Points

- Pure java implementation
- Allows method interception
  - No field or property intercepts yet
- AOP advice is specified using typical bean definitions
  - Closely integrates with Spring IoC
- Proxy based AOP
  - J2SE dynamic proxies or CGLIB proxies
- Not a replacement for AspectJ

3.6.3) Why AOP?

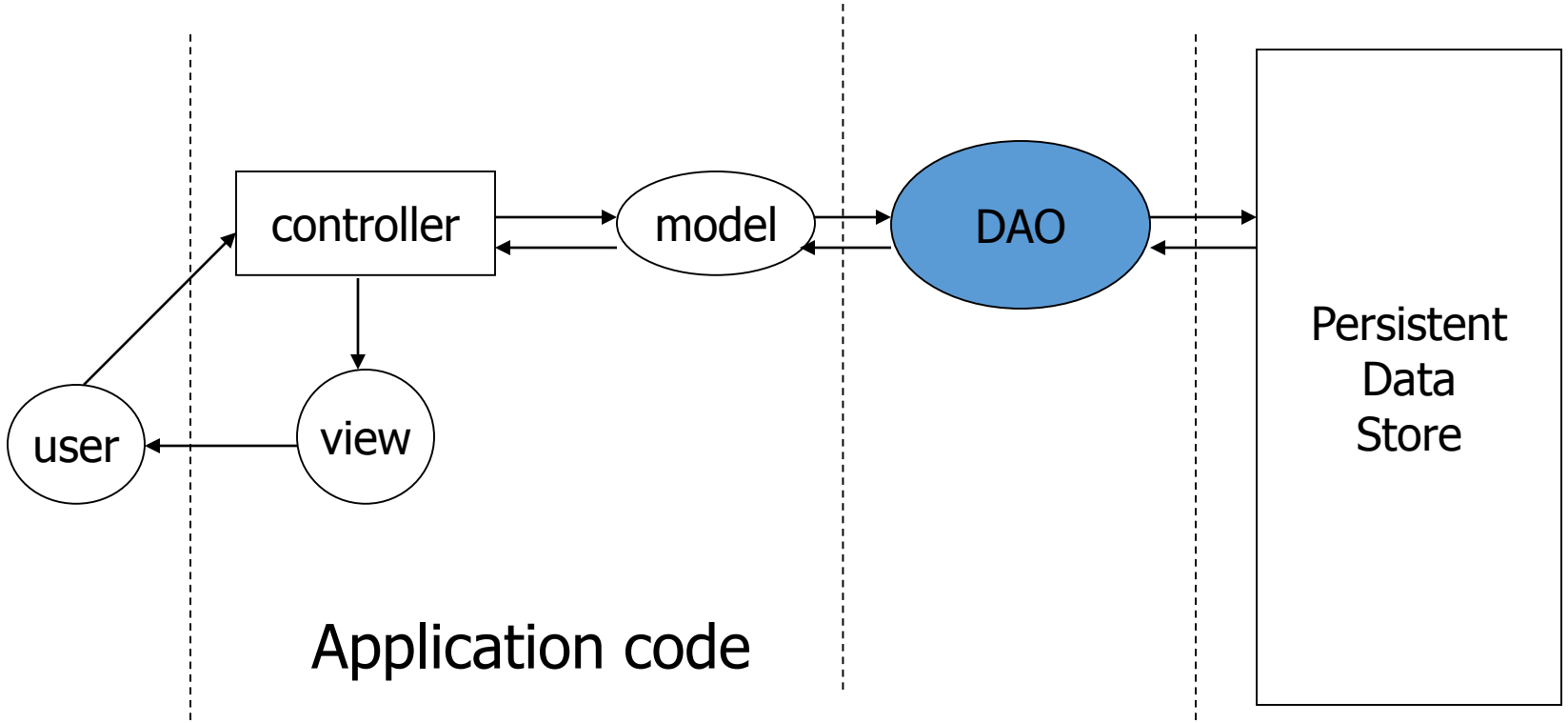
- **AOP complements IoC to deliver a non-invasive framework**
- Externalizes *crosscutting concerns* from application code
  - Concerns that cut across the structure of an object model
  - AOP offers a different way of thinking about program structure to an object hierarchy
- EJB interception is conceptually similar, but not extensible and imposes too many constraints on components
- Spring provides important out-of-the box aspects
  - Declarative transaction management for any POJO
  - Pooling
  - Resource acquisition/release

3.7) AOP + IOC: A UNIQUE SYNERGY

- AOP + IoC is a match made in heaven
- Any object obtained from a Spring IoC container can be transparently advised based on configuration
- Advisors, pointcuts and advices can themselves be managed by the IoC container
- *Spring is an integrated, consistent solution*

3.8) Data Access Object (DAO)...

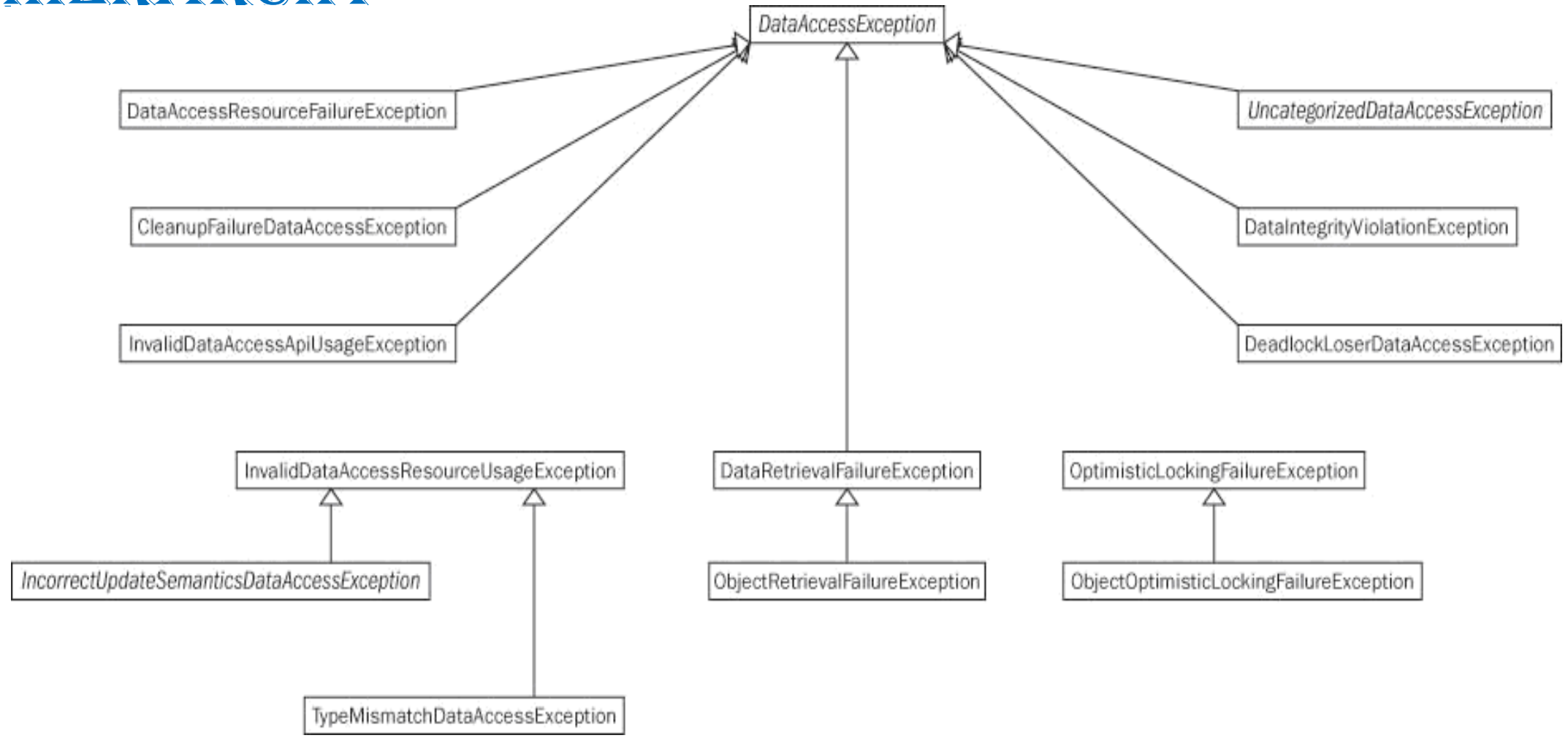
- A Java EE design pattern



3.8) Spring DAO

- Integrated with Spring transaction management
  - Unique synergy
  - Gestalt again…
- Doesn't reinvent the wheel.
  - *There are good solutions for O/R mapping, we make them easier to use*
- Out-of-the-box support for
  - JDBC
  - Hibernate
  - JDO
  - iBATIS
- Model allows support for other technologies (TopLink etc)
- Consistent `DataAccessException` hierarchy allows truly technology-agnostic DAOs

3.8.1) Spring DAO: Consistent Exception Hierarchy



4) Spring Bean Config File.

4.1) What is a Bean?...

- Typical java bean with a unique id
- In spring there are basically two types
  - Singleton
    - One instance of the bean created and referenced each time it is requested
  - Prototype (non-singleton)
    - New bean created each time
    - Same as **new** ClassName()
- Beans are normally created by Spring as late as possible

4.1) What is a Bean?

- Defines a bean for Spring to manage
  - Key attributes
    - class (required): fully qualified java class name
    - id: the unique identifier for this bean
    - *configuration*: (singleton, init-method, etc.)
    - constructor-arg: arguments to pass to the constructor at creation time
    - property: arguments to pass to the bean setters at creation time
    - Collaborators: other beans needed in this bean (a.k.a dependencies), specified in property or constructor-arg
- Typically defined in an XML file

4.1) Spring Bean Definition

- The bean class is the actual implementation of the bean being described by the BeanFactory.
- Bean examples – DAO, DataSource, Transaction Manager, Persistence Managers, Service objects, etc
- Spring config contains implementation classes while your code should program to interfaces.
- Bean behaviors include:
  - Singleton or prototype
  - Autowiring
  - Initialization and destruction methods
    - init-method
    - destroy-method
- Beans can be configured to have property values set.
  - Can read simple values, collections, maps, references to other beans, etc.

4.2) Spring Bean Factory

- Bean Factory is core to the Spring framework
  - Lightweight container that loads bean definitions and manages your beans.
  - Configured declaratively using an XML file, or files, that determine how beans can be referenced and wired together.
  - Knows how to serve and manage a singleton or prototype defined bean
  - Responsible for lifecycle methods.
  - Injects dependencies into defined beans when served
- Avoids the use of singletons and factories

4.3.1) Simple Spring Bean Example

```
<bean id="orderBean" class="example.OrderBean"
      init-method="init">
      <property
      name="minimumAmountToProcess">10</property>
      <property name="orderDAO">
        <ref bean="orderDAOBean"/>
      </property>
</bean>
```

```
public class OrderBean implements IOrderBean{
      …
      public void
      setMinimumAmountToProcess(double d){
            this. minimumAmountToProcess = d;
      }
      public void setOrderDAO(IOrderDAO odao){
            this.orderDAO = odao;
      }
}
```

4.3.2) Sample Bean Definition

```
<bean id="exampleBean" class=”org.example.ExampleBean">
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>
  <property name="beanTwo"><ref bean="yetAnotherBean"/></property>
  <property name="integerProperty"><value>1</value></property>
</bean>
```

```
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; }
        …
}
```

4.3.3) How are beans created?

- Beans are created in order based on the dependency graph
  - Often they are created when the factory loads the definitions
  - Can override this behavior in bean

    <bean class="“className” lazy-init="“true” />
  - You can also override this in the factory or context but this is not recommended
- Spring will instantiate beans in the order required by their dependencies
  1. app scope singleton - eagerly instantiated at container startup
  2. lazy dependency - created when dependent bean created
  3. VERY lazy dependency - created when accessed in code

4.3.4) How are beans injected?

- A dependency graph is constructed based on the various bean definitions
- Beans are created using constructors (mostly no-arg) or factory methods
- Dependencies that were not injected via constructor are then injected using setters
- Any dependency that has not been created is created as needed

4.4) Multiple Bean Config Files

- There are 3 ways to load multiple bean config files (allows for logical division of beans)
  - Load multiple config files from web.xml

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>classpath:/WEB-INF/spring-config.xml, classpath:/WEB-
  INF/applicationContext.xml</param-value>
</context-param>
```

  - Use the import tag

```
<import resource="services.xml"/>
```

  - Load multiple config files using Resources in the application context constructor
    - Recommended by the spring team
    - Not always possible though

```
ClassPathXmlApplicationContext appContext = new
  ClassPathXmlApplicationContext( new String[]
  {"applicationContext.xml", "applicationContext-
  part2.xml"});
```

4.5) Bean Properties?

- The primary method of dependency injection
- Can be another bean, value, collection, etc.

```
<bean id="exampleBean" class="org.example.ExampleBean">
    <property name="anotherBean">
        <ref bean="someOtherBean" />
    </property>
</bean>
```

- This can be written in shorthand as follows

```
<bean id="exampleBean" class="org.example.ExampleBean">
    <property name="anotherBean" ref="someOtherBean" />
</bean>
```

4.5.1) Anonymous Vs ID

- Beans that do not need to be referenced elsewhere can be defined anonymously
- This bean is identified (has an id) and can be accessed to inject it into another bean

```
<bean id="exampleBean" class="org.example.ExampleBean">
    <property name="anotherBean" ref="someOtherBean" />
</bean>
```

- This bean is anonymous (no id)

```
<bean class="org.example.ExampleBean">
    <property name="anotherBean" ref="someOtherBean" />
</bean>
```

4.5.2) What is an inner bean?

```
<bean id="outer" class="org.example.SomeBean">
    <property name="person">
        <bean class="org.example.PersonImpl">
            <property name="name"><value>Aaron</value></property>
            <property name="age"><value>31</value></property>
        </bean>
    </property>
</bean>
```

- It is a way to define a bean needed by another bean in a shorthand way
  - Always anonymous (id is ignored)
  - Always prototype (non-singleton)

4.5.3) Bean init-method

- The init method runs AFTER all bean dependencies are loaded
  - Constructor loads when the bean is first instantiated
  - Allows the programmer to execute code once all dependencies are present

```
<bean id="exampleBean" class=”org.example.ExampleBean"
        init-method=”init” />
```

```
public class ExampleBean {
    public void init() {
        // do something
    }
}
```

4.5.4) Bean Values

- Spring can inject more than just other beans
- Values on beans can be of a few types
  - Direct value (string, int, etc.)
  - Collection (list, set, map, props)
  - Bean
  - Compound property

Example of injecting a string value

```
<bean class="org.example.ExampleBean">
   <property name="email">
            <value>azeckoski@gmail.com</value>
   </property>
</bean>
```

4.5.5) Abstract (Parent) Beans

- Allows definition of part of a bean which can be reused many times in other bean definitions

```
<bean id="abstractBean" abstract="true"
            class="org.example.ParentBean">
   <property name="name" value="parent-AZ"/>
   <property name="age" value="31"/>
</bean>

<bean id="childBean"
            class="org.example.ChildBean"
            parent="abstractBean" init-method="init">
   <property name="name" value="child-AZ"/>
</bean>
```

- *The parent bean defines 2 values (name, age)*
- *The child bean uses the parent age value (31)*
- *The child bean overrides the parent name value (from parent-AZ to child-AZ)*
- *Parent bean could not be injected, child could*

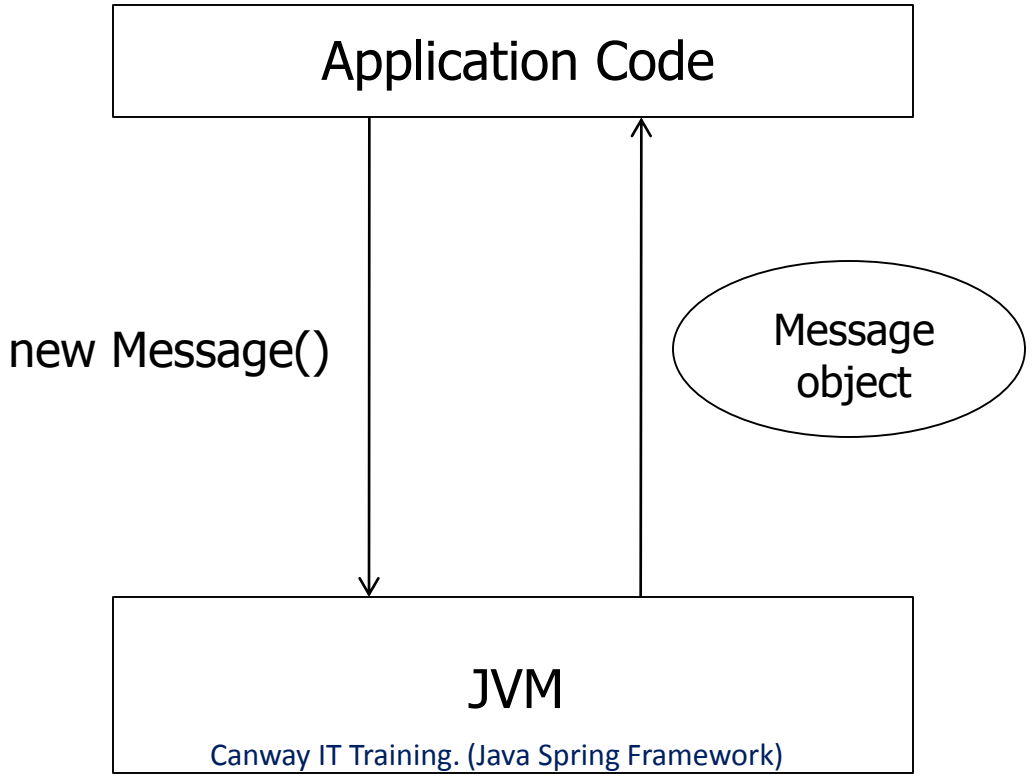
4.6) Bean Configuration File

```
<beans>

   <bean id="msgBean“
         class=“cs520.spring.hello.Message">
      <property name="message" value="Hello World!" />
   </bean>

</beans>
```

- The string “`Hello World`” is injected to the bean `msgBean`

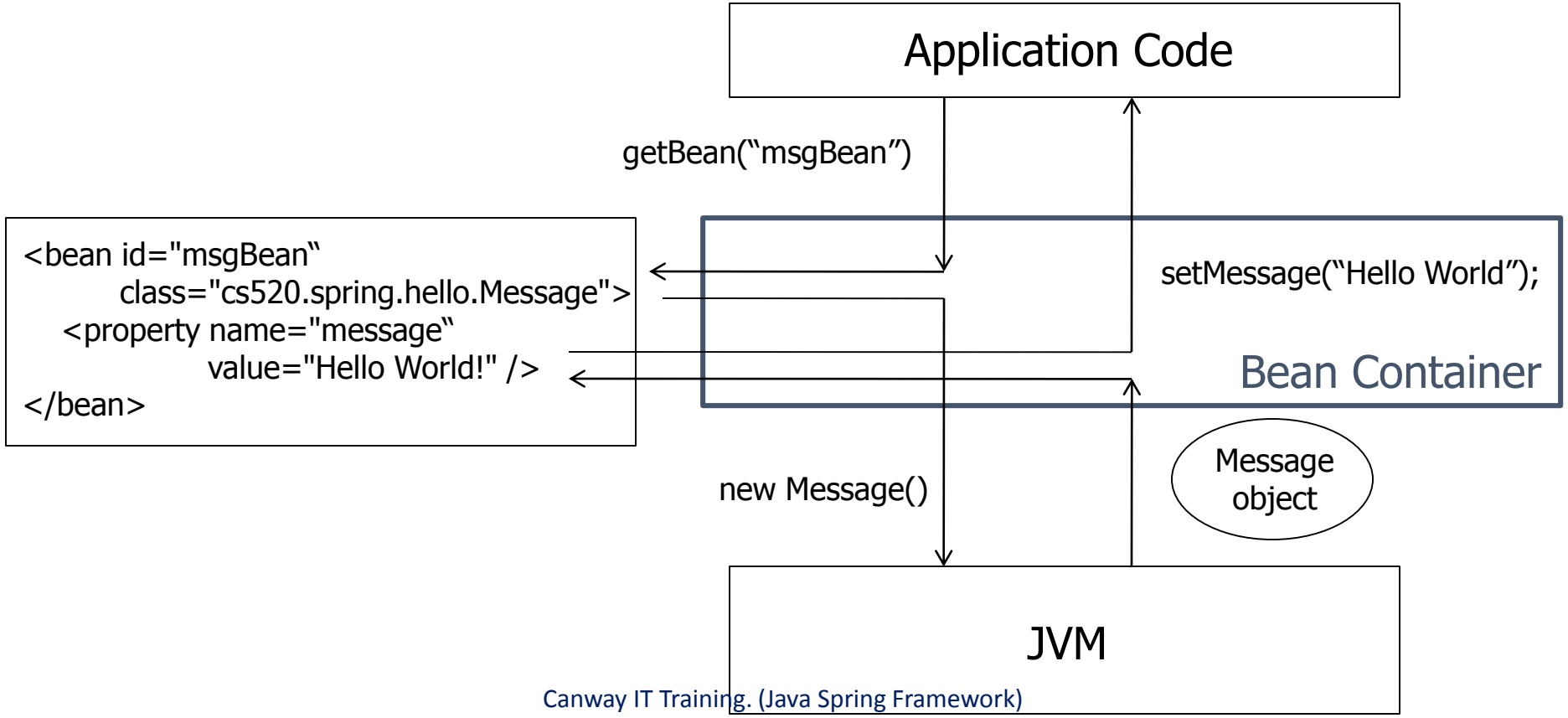
4.6.1) Understand Bean Container ...

- Without a bean container



4.6.1) Understand Bean Container

- With a bean container



4.6.2) Quick Summary of Bean Configuration

Bean	<bean>, “id”, “class”
Simple type property	<property>, “name”, “value”
Class type property	<property>, “name”, “ref” (to another <bean>)
Collection type property	<list>/<set>/<map>/<props>, <value>/<ref>/<entry>/<prop>
Constructor arguments	<constructor-arg>, “index”, same as other properties

4.6.3) Some Bean Configuration Examples

```
<property name="foo">
    <set>
        <value>bar1</value>
        <ref bean="bar2" />
    </set>
</property>
```

```
<property name="foo">
    <map>
        <entry key="key1">
            <value>bar1</value>
        </entry>
        <entry key="key2">
            <ref bean="bar2" />
        </entry>
    </map>
</property>
```

```
<property name="foo">
    <props>
        <prop key="key1">bar1</prop>
        <prop key="key2">bar2</prop>
    </props>
</property>
```

4.6.4) Annotation-based configuration

- Activate annotation processing with `<context:annotation-config />`
- Automatically scan for Spring bean with `<context:component-scan />`
- Mark a class to be a Spring bean with `@Component`
- Enable auto wiring with `@Autowired`

4.7) XML NAMESPACE ...

```
<?xml version="1.0" encoding="UTF-8"?>
<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:annotation-config />

    <context:component-scan base-package="cs520.spring.stack" />

</beans>
```

4.7) XML NAMESPACE

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns=" http://www.springframework.org/schema/context  "
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    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">

    <annotation-config />

    <component-scan base-package="cs520.spring.stack" />

</bean:beans>
```

5) Code examples.

5.1) Code Example 1...

- What's wrong here?

```
public class Mechanic {
  public void fixCar() {
    PhillipsScrewdriver tool =
          new PhillipsScrewdriver();
    tool.use();
  }
}
```

How about now?

```
public class Mechanic {
  public void fixCar() {
    Tool tool =
          new PhillipsScrewdriver();
    tool.use();
  }
}
```

5.1) Code Example 1...

- Any better?

```
public class Mechanic {
  public void fixCar() {
    ToolFactory tf =
          ToolFactory.getInstance();
    Tool tool = tf.getTool();
    tool.use();
  }
}
```

- Certainly this is better...

```
public class Mechanic {
  public void fixCar() {
    InitialContext ctx = null;
    try {
      ctx = new InitialContext();
      Tool quest = (Tool) ctx.lookup(
          "java:comp/env/Tool");
      tool.use();
    } catch (NamingException e) {
    } finally {
      if(ctx != null) {
        try {ctx.close(); }
        catch (Exception e) {}
      }
    }
  }
}
```

5.1) Code Example 1

- Let's try again...

```
public class Mechanic {
  private Tool tool;
  public Mechanic(Tool tool) {
    this.tool = tool;
  }

  public void fixCar() {
    tool.use();
  }
}
```

- Or maybe this...

```
public class Mechanic {
  private Tool tool;
  public void setTool(Tool tool) {
    this.tool = tool;
  }

  public void fixCar() {
    tool.use();
  }
}
```

5.2) Dependency Injection

- Objects are given what they need
- Coupling is low when used with interfaces
- Makes classes easier to swap out
- Makes classes easier to unit test

- DI in Spring
- Several options
  - XML
  - Annotation-driven
  - Java-based configuration
- None are mutually exclusive

5.3.1) XML-based Wiring

- <bean id="screwdriver"
- class="com.habuma.tools.PhillipsScrewdriver" />

- <bean id="mechanic"
- class="com.habuma.mechanic.AutoMechanic">
- <constructor-arg ref="screwdriver" />
- </bean>

```
<bean id="screwdriver"
    class="com.habuma.tools.PhillipsScrewdriver" />

<bean id="mechanic"
    class="com.habuma.mechanic.AutoMechanic">
  <property name="tool" ref="screwdriver" />
</bean>
```

5.3.2) Annotation-Based Wiring

```
<context:component-scan
   base-package="com.habuma.mechanic" />
```

```
public class Mechanic {
  private Tool tool;
@Autowired
  public void setTool(Tool tool) {
    this.tool = tool;
  }

  public void fixCar() {
    tool.use();
  }
}
```

- public class Mechanic {
  - @Autowired
- private Tool tool;

- public void fixCar() {
- tool.use();
- }
- }

5.4) Java-Based Configuration

```
@Configuration
public class AutoShopConfig {
  @Bean
  public Tool screwdriver() {
    return new PhillipsScrewdriver();
  }

  @Bean
  public Mechanic mechanic() {
    return new AutoMechanic(screwdriver());
  }
}
```

5.5) Life Without AOP

```
public void withdrawCash(double amount) {
   UserTransaction ut = context.getUserTransaction();

   try {
      ut.begin();
      updateChecking(amount);
      machineBalance -= amount;
      insertMachine(machineBalance);
      ut.commit();
   } catch (ATMException ex) {
       LOGGER.error("Withdrawal failed");
       try {
          ut.rollback();
       } catch (SystemException syex) {
          // ...
       }
    }
}
```

- Life with AOP

```
public void withdrawCash(double amount) {
   try {
      updateChecking(amount);
      machineBalance -= amount;
      insertMachine(machineBalance);
   } catch (ATMException ex) {
      LOGGER.error("Withdrawal failed");
   }
}
```

5.6) Logging Aspect

```
public class LoggingAspect {
  private static final Logger LOGGER =
      Logger.getLogger(LoggingAspect.class);
  public logBefore() {
    LOGGER.info("Starting withdrawal");
  }

  public logAfterSuccess() {
    LOGGER.info("Withdrawal complete");
  }

  public logFailure() {
    LOGGER.info("Withdrawal failed");
  }
}
```

5.7) XML-BASED AOP

```
<bean id="loggingAspect"
    class="LoggingAspect" />

<aop:config>
  <aop:aspect ref="loggingAspect">
    <aop:before
        pointcut="execution(* *.withdrawCash(..))"
        method="logBefore" />
    <aop:after-returning
        pointcut="execution(* *.withdrawCash(..))"
        method="logBefore" />
    <aop:after-throwing
        pointcut="execution(* *.withdrawCash(..))"
        method="logBefore" />
  </aop:aspect>
</aop:config>
```

5.8) Annotation-based AOP

```
@Aspect
public class LoggingAspect {
  private static final Logger LOGGER =
      Logger.getLogger(LoggingAspect.class);
  @Pointcut("execution(* *.withdrawCash(..))")
  public void withdrawal() {}

  @Before("withdrawal()")
  public logBefore() {
    LOGGER.info("Starting withdrawal");
  }
  @AfterReturning("withdrawal()")
  public logAfterSuccess() {
    LOGGER.info("Withdrawal complete");
  }

  @AfterThrowing("withdrawal()")
  public logFailure() {
    LOGGER.info("Withdrawal failed");
  }
}
```

```
<aop:aspectj-autoproxy />
```

6) SUMMARY.

SPRING...

- Is a lightweight container framework
- Simplifies Java development
- POJO-oriented
- Promotes loose coupling with dependency injection
- Supports declarative programming with AOP
- Eliminates boilerplate code with templates

How does Spring do it?

- Spring controls instantiation of all the objects and passes a Spring reference to these objects using various DI – Dependency Injection techniques like getter/setter injection, and constructor injection (aka IoC - Inversion of Control).

AOP (Aspect Oriented Programming)...

- One of the key components of Spring Framework is the Aspect oriented programming (AOP) framework.
- Aspect-Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic.
- Common good examples of aspects: logging, auditing, declarative transactions, security, caching, etc.

AOP TERMINOLOGIES

1	**Aspect** This is a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement.
2	**Join point** This represents a point in your application where you can plug-in the AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.
3	**Advice** This is the actual action to be taken either before or after the method execution. This is an actual piece of code that is invoked during the program execution by Spring AOP framework.
4	**Pointcut** This is a set of one or more join points where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples.
5	**Introduction** An introduction allows you to add new methods or attributes to the existing classes.
6	**Target object** The object being advised by one or more aspects. This object will always be a proxied object, also referred to as the advised object.
7	**Weaving** Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime.

TYPES OF AOP ADVICE

1	**before** Run advice before the a method execution.
2	**after** Run advice after the method execution, regardless of its outcome.
3	**after-returning** Run advice after the a method execution only if method completes successfully.
4	**after-throwing** Run advice after the a method execution only if method exits by throwing an exception.
5	**around** Run advice before and after the advised method is invoked.