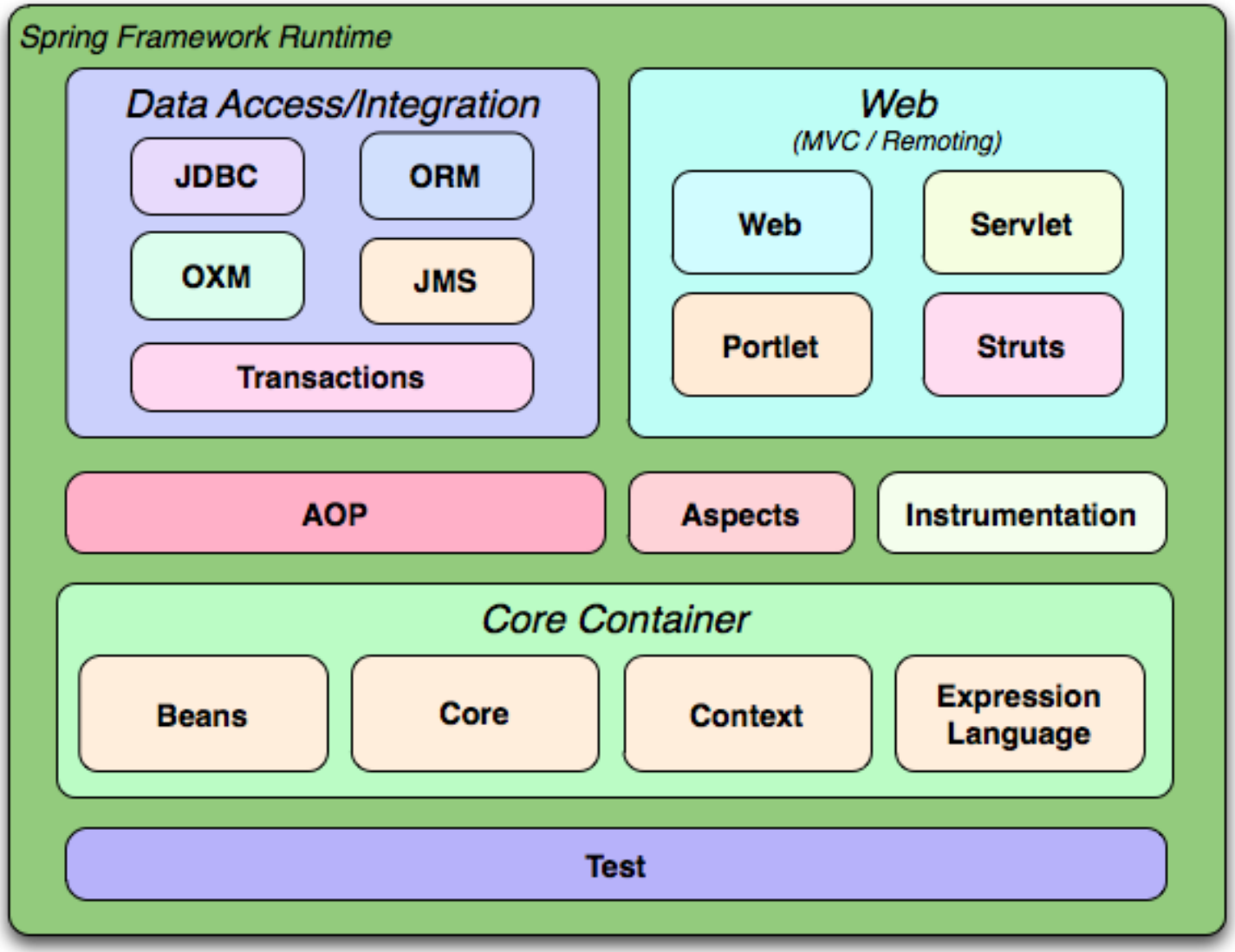
# Spring Framework Basics


## INF5750/9750 - Lecture 2 (Part I)

# Spring framework - Core and modules

- A Java platform that is used to build n-tier applications
- Incorporates many design patterns that enable application code to follow “best practices”
  - Factory pattern
  - MVC pattern
  - Decorator pattern
  - …
- Different, but yet complementary to the JavaEE world
- Spring Framework can be broadly divided into
  - Core Container
  - Data Access/Integration
  - Web
  - AOP & Instrumentation
  - Test

# Spring overview



# Logging and other dependencies

- Different logging libraries as choice with Spring, but Apache JCL is a mandatory dependency with Spring
- JDK JUL, slf4j, log4j… can be used with Spring

- No big blob .jar file for all spring core+modules
  - Separate dependencies - spring-*.jar
  - spring-core-**.RELEASE.jar
  - spring-webmvc-**.RELEASE.jar
  - spring-aop-**.RELEASE.jar

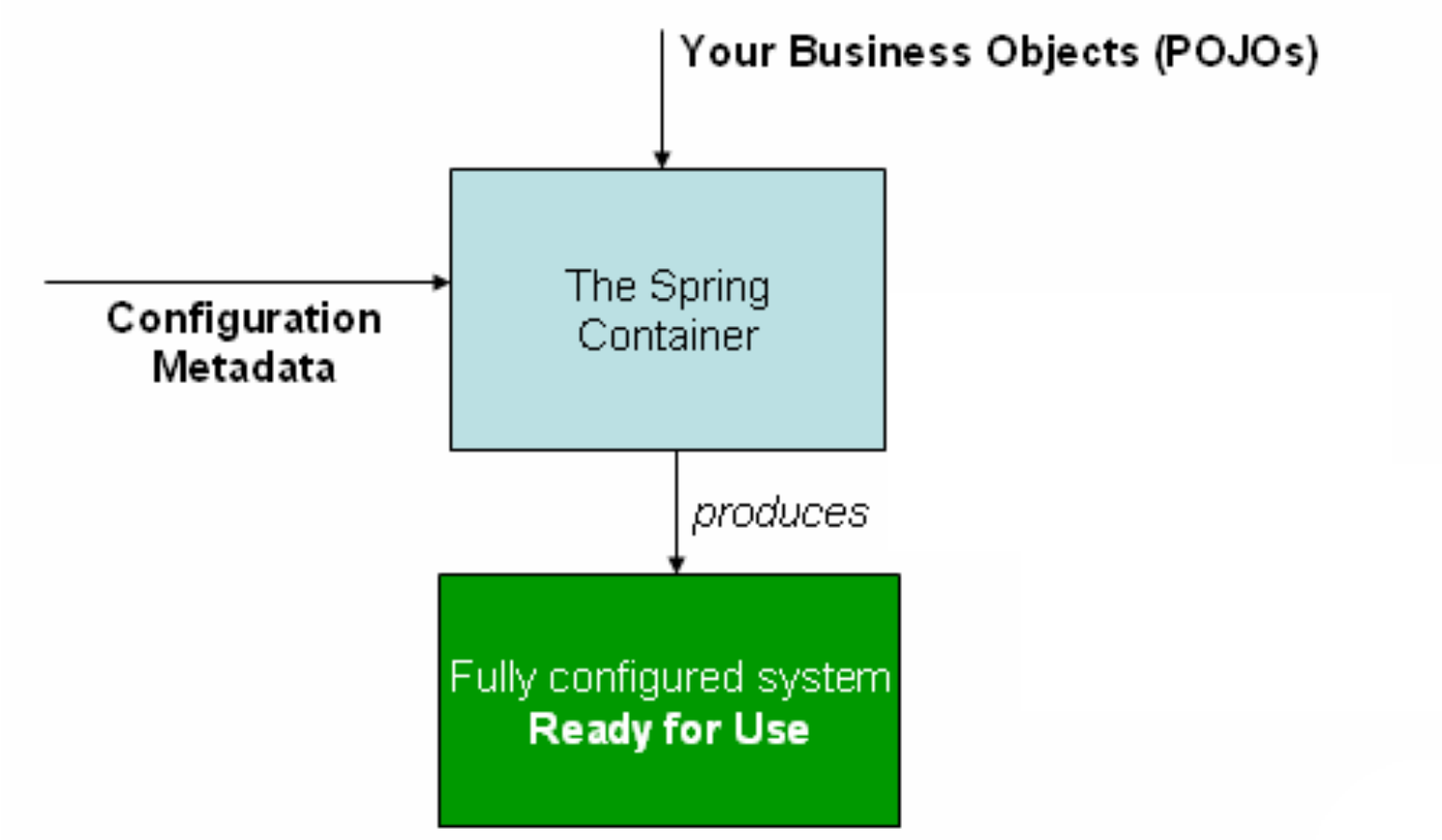
- Other related projects from Spring
  - Spring Security, Spring Web Flow, Spring Data … etc
  - e.g. spring-security-*-**.RELEASE.jar (sub-modules)

# The IoC container

- “Inversion of Control”, a generic term to describe that the bean does not control its instantiation or location of its dependency (like Service Locator pattern)
- The IoC container is the core component of the Spring framework
- A bean is an object that is managed by the IoC container
- The IoC container is responsible for instantiating, assembling and managing beans
- Spring comes with two types of containers
  - BeanFactory
  - ApplicationContext

- TIP: Using a single IoC container is expected

# The IoC container



# The BeanFactory

- Provides basic support for dependency injection
- Responsible for
  - Creating and dispensing beans
  - Managing dependencies between beans
- Lightweight – useful when resources are scarce
  - Mobile applications, applets

- XMLBeanFactory most commonly used implementation

```
Resource xmlFile = new ClassPathResource( "META-INF/beans.xml" );

BeanFactory beanFactory = new XmlBeanFactory( xmlFile );
```

```
MyBean myBean = (MyBean) beanFactory.getBean( ”myBean” );
```

# The ApplicationContext

- Built on top of the BeanFactory
- Provides more enterprise-centric functionality
  - Internationalization, AOP, transaction management
  - Preferred over the BeanFactory in most situations.
  - Use ApplicationContext, unless you are integrating with low-level frameworks

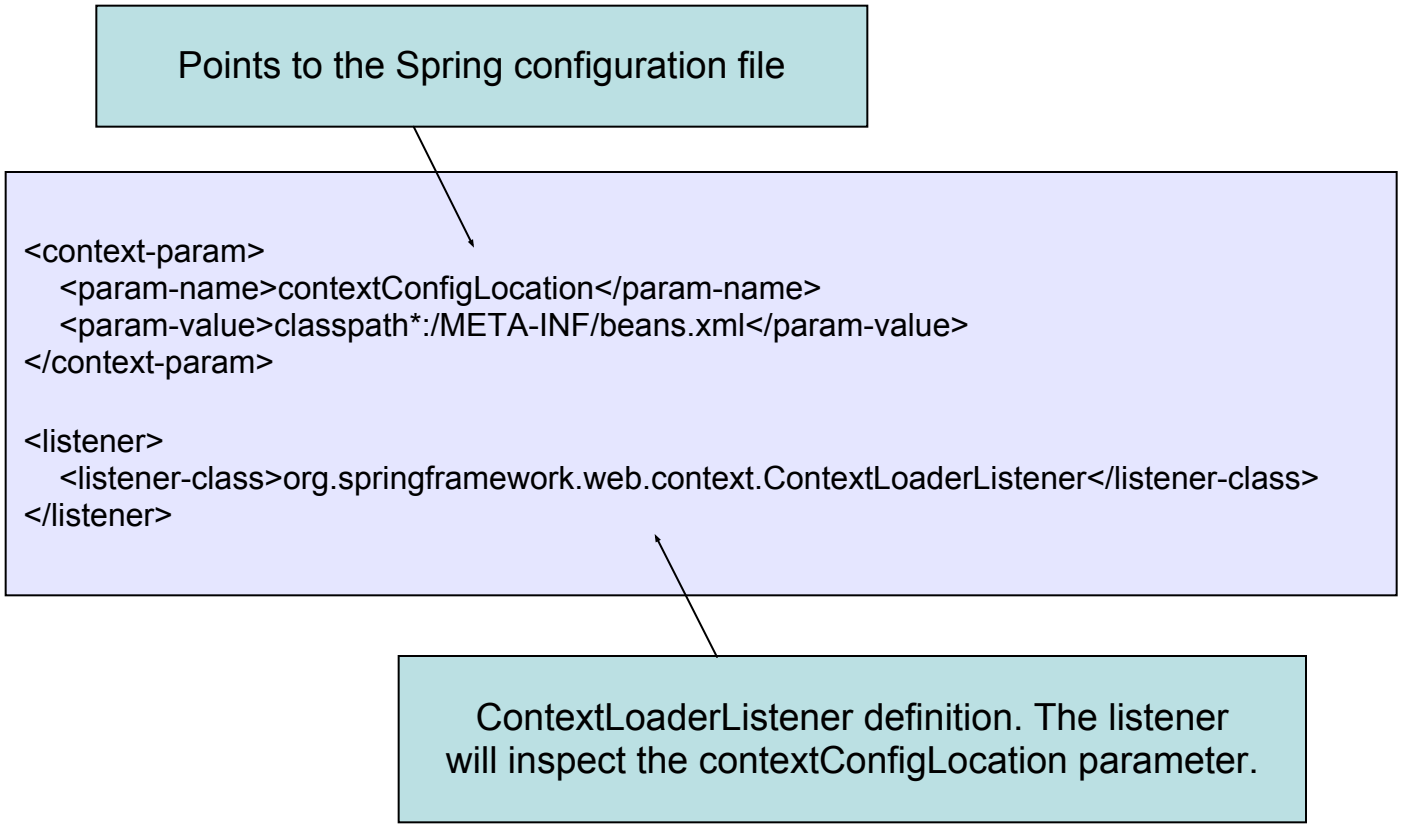
- Most commonly used implementation is the ClassPathXmlApplicationContext

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("classpath*:conf/appContext.xml");
```

```
MyBean myBean = (MyBean) context.getBean( ”myBean” );
```

# Convenient container instantiation

ApplicationContext instances can be created declaratively in web.xml using a ContextLoader



# Dependencies

- An application consists of many beans working together
- Dependency: a bean being used by another bean

```
public class DefaultStudentSystem implements StudentSystem
{
   private String studentDAO;

   public void setStudentDAO( StudentDAO studentDAO )
   {
      this.studentDAO = studentDAO;
   }
```

Dependency defined only through a reference and a public set-method

```
<bean id="studentSystem"  class="no.uio.inf5750.service.DefaultStudentSystem">
   <property name="studentDAO" ref bean="studentDAO"/>
</bean>

<bean id="studentDAO" class="no.uio.inf5750.dao.HibernateStudentDAO"/>
```

StudentDAO injected into StudentSystem

- **We discuss Dependency Injection in detail in next presentation**

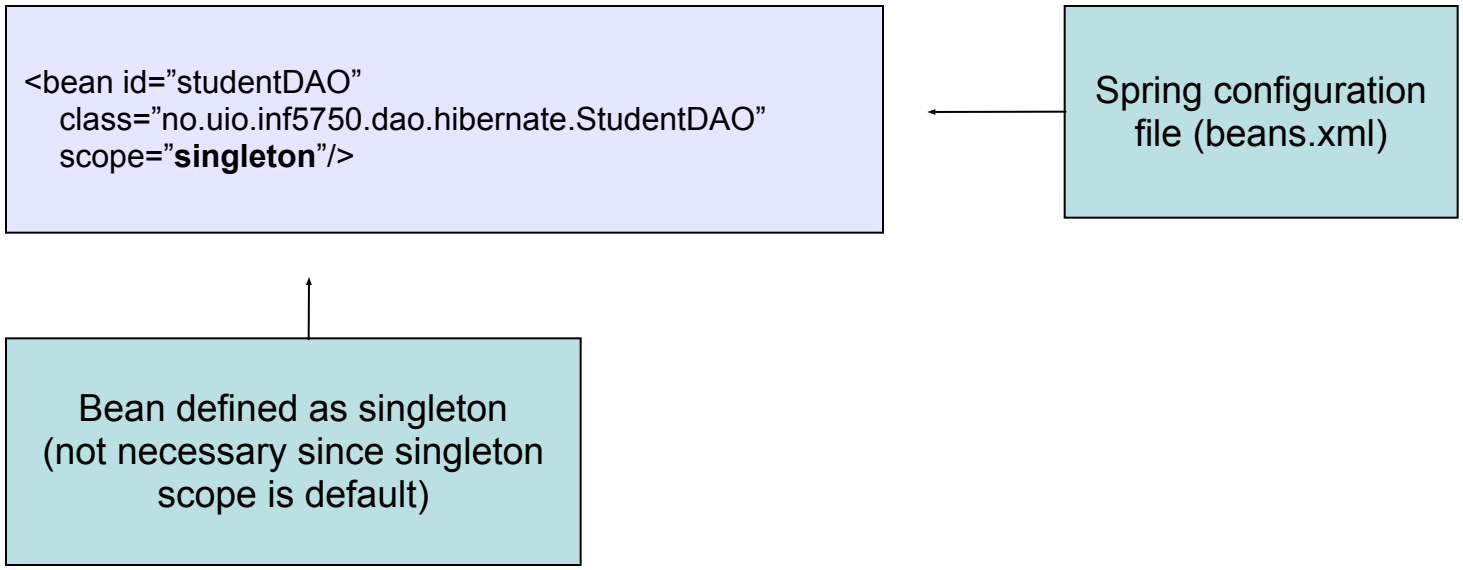
# Bean scopes

- A bean definition is a recipe for creating instances
  - Many object instances can be created from a single definition
- Spring can manage the scope of the beans
  - No need for doing it programmatically

| *Scope* | *Description* |
| --- | --- |
| singleton | Scopes a single bean definition to a single object instance. |
| prototype | Scopes a single bean definition to any number of object instances. |

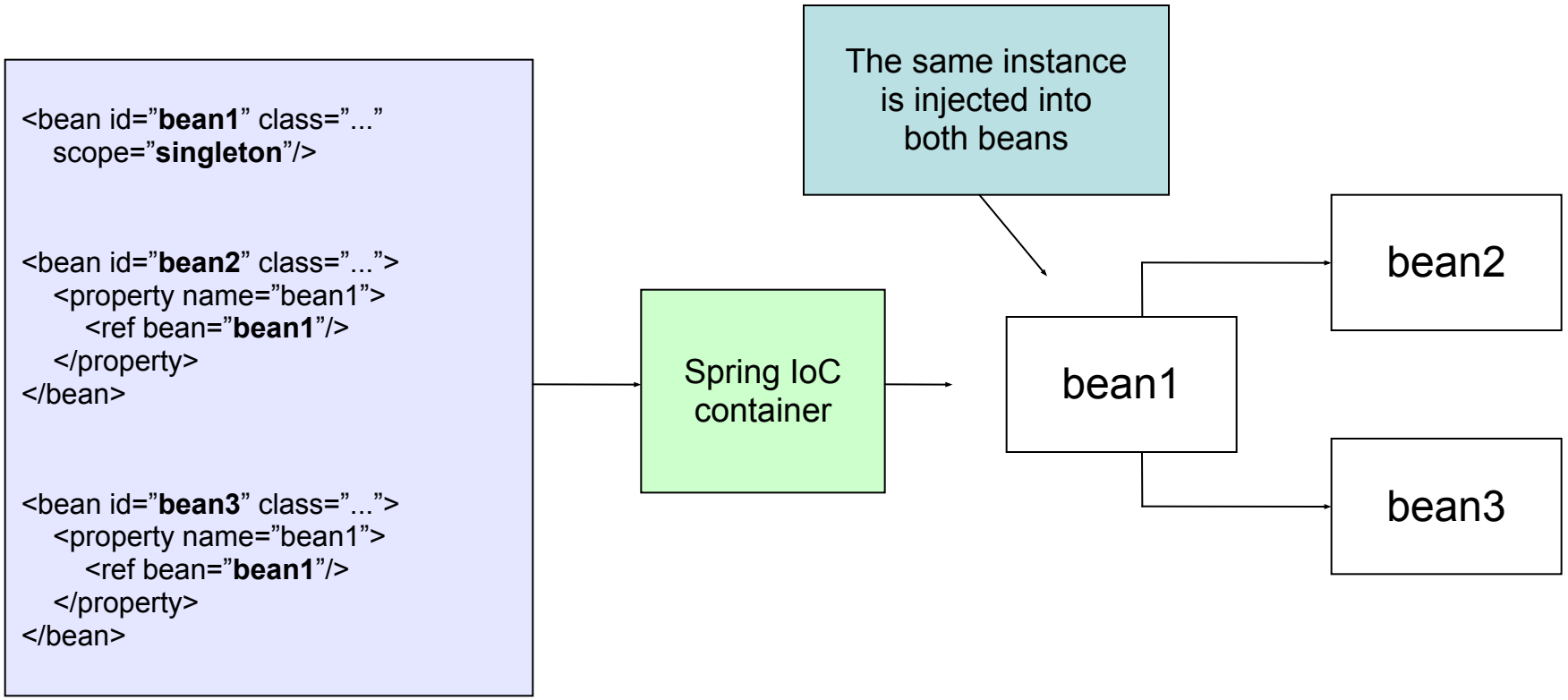
# The singleton scope

- Only one shared instance will ever be created by the container
- The single bean instance will be stored in a cache and returned for all requests
- Singleton beans are created at container startup-time



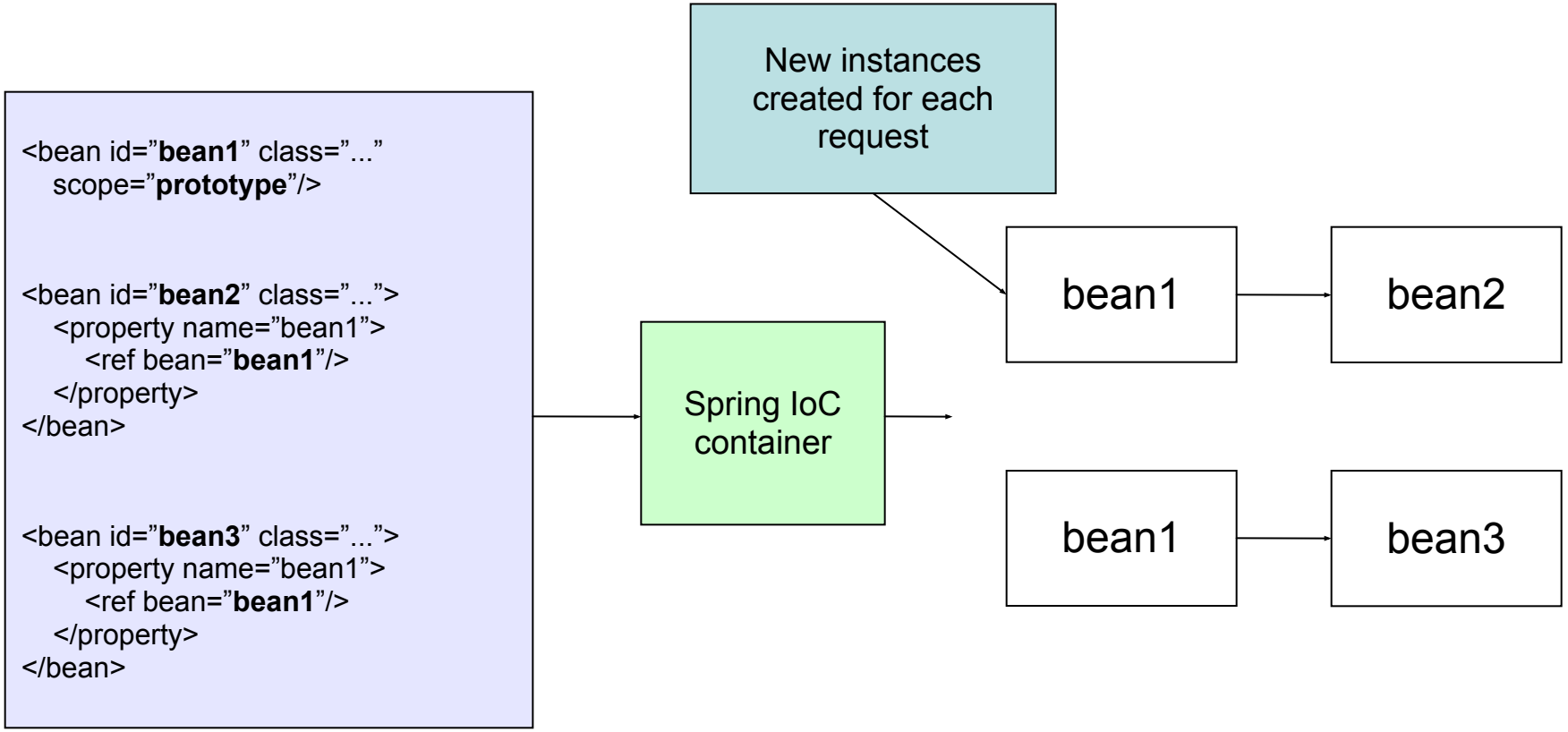
# The singleton scope

- Singleton per container – not by classloader
- Singleton is default scope in Spring



# The prototype scope

- A new bean instance will be created for each request
- Use prototype scope for stateful beans
- Use singleton scope for stateless beans



# Customizing the lifecycle of a bean

- Spring lets you define callback methods which are invoked at bean initialization and destruction
- The *init* method will be invoked after all properties are set on the bean
- The *destroy* method will be invoked when the container containing the bean is destroyed (not prototypes)

```
<bean id="lifecycleBean"
   class=" no.uio.inf5750.example.spring.lifecycle.LifecycleBean"
   init-method="init"/>
```

Spring configuration file

```
public class LifecycleBean
{
   public void init()
   {
      // do something useful initialization work
   }
   public void destroy()
   {
      // do some useful destruction work
   }
}
```

Java bean

# Annotation-based container configuration

- IoC container can ALSO be configured using *Annotations*
  - Configuration closer to code → More contextual
  - Shorter and more concise configuration
  - No clear separation of code and configuration
  - Decentralized configuration and harder to control

- Annotation injection is performed *before* XML injection. So XML injection overrides

To enable Annotation-based configuration of applicationContext →

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
…>
    <context:annotation-config/>

</beans>
```

```
@Scope("prototype")
@Component
public class DefaultStudentSystem implements StudentSystem
{
   @Autowired
   private StudentDAO studentDAO;
…
```

Many different annotations to configure beans, contexts etc.

# Spring 3.2.x Supported Annotations

- @Required - Indicates that the bean must be populated
- @Autowired
  - Methods can be autowired to inject beans used as parameters
  - Constructors and Fields can be autowired as well
  - All beans of a type can be autowired (Student[] students)
- @Component - a generic stereotype for any Spring-managed
- @Repository - Data Access Object (DAO)
- Other popular ones: @Service, @Controller, @Bean, … etc.

```
@Component
public class DefaultStudentSystem implements StudentSystem
{
   @Autowired (required=false)
   private StudentDAO studentDAO;

  @Autowired
   public void enrollStudentsToCourse( Student[] student, Course c )
   { …..
```

*required* attribute is preferred over @Required

# Other Annotations

- Dependency Injection using JSR330 - **@Inject** and **@Named**

```
@Named("studentSystem")
public class DefaultStudentSystem implements StudentSystem {
    private String studentDAO;

    @Inject
    public void setStudentDAO( StudentDAO studentDAO )
    {
        this.studentDAO = studentDAO;
    }
```

*@Qualifier* is used in Spring in place of JSR330's @Named

- The JSR-250 **@PostConstruct** and **@PreDestroy** annotations are generally considered best practice for receiving lifecycle callbacks in a modern Spring application

# Annotation config instead of XML

- In JavaSE applications

```
public static void main(String[] args) {
  AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
  ctx.scan("uio.no");
  ctx.refresh();
  TransferService transferService = ctx.getBean(TransferService.class);
}
```



```
<beans>
  <context:component-scan
        base-package="uio.no"
/>
</beans>
```

- In web apps, the web.xml, you do the following

```
<web-app>
  <context-param>
     <param-name>contextClass</param-name>
     <param-value>
        org.springframework.web.context.support.AnnotationConfigWebApplicationContext
     </param-value>
  </context-param>
  <context-param>
     <param-name>contextConfigLocation</param-name>
     <param-value>uio.no.AppConfig</param-value>
  </context-param>
….
</web-app>
```

```
@Configuration
public class AppConfig {
  @Bean
  public TransferService transferService() {
     return new TransferServiceImpl();
  }
```

In XML config



```
<beans>
  <bean id="transferService"
          class="uio.no.TransferServiceImpl"/>
</beans>
```

# Internationalization

- Internationalization (i18n) is the process of decoupling the application from any specific locale
- Makes it possible to display messages in the user's native language
- The ApplicationContext extends the MessageSource interface which provides i18n functionality
- Spring also provides the interface HierarchicalMessageSource, which can resolve messages hierarchically
- When an ApplicationContext is loaded, it automatically searches for a MessageSource bean defined in the context. The bean must have the name ***messageSource***
- Most commonly used implementation is the provided ResourceBundleMessageSource

# The SaluteService

```
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="i18n"/>
</bean>

<bean id="saluteService"
    class="no.uio.inf5750.example.spring.i18n.DefaultSaluteService">
    <property name="messages" ref="messageSource"/>
</bean>
```

Spring looks for a bean called *messageSource*

Basename for resourcebundles to use. e.g. Spring will look for ***i18n.properties*** on classpath

MessageSource injected into DefaultSaluteService

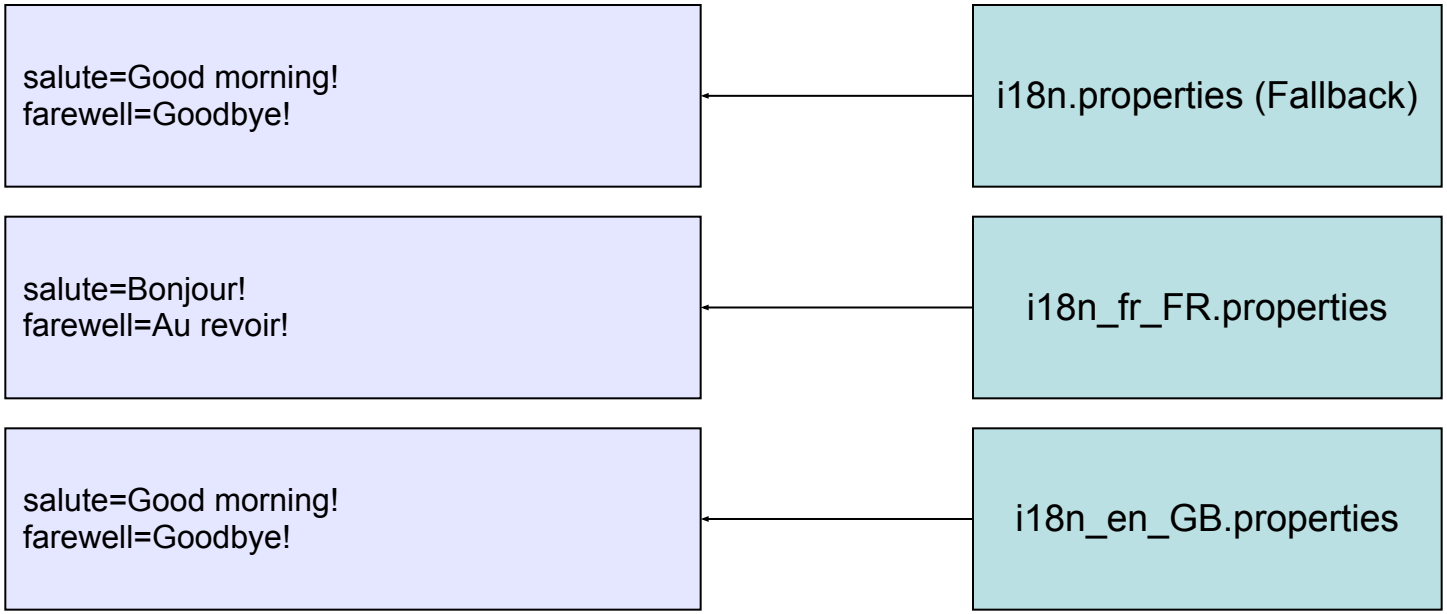
```
public class DefaultSaluteService implements SaluteService
{
    private MessageSource messages;

    // set-method for messages

    public String salute()
    {
        return messages.getMessage( "salute", null, locale );
    }
```

getMessage is invoked

*param1:* property key
*param2:* arguments
*param3:* Locale

# The SaluteService

- MessageResource follows the the locale resolution and fallback rules of the standard JDK ResourceBundle



# Application Events / Listeners

- Events can be published to the ApplicationContext

- Bean implements *ApplicationListener,* it gets notified when ApplicationEvents are published to ApplicationContext
  - Observer pattern
  - Custom event needs to extend **ApplicationEvent** class

- Spring standard events
  - ContextRefreshedEvent, ContextStoppedEvent, RequestHandledEvent etc.

- Custom events published to ApplicationContext by calling
  - ***publishEvent(Event)*** on ApplicationEventPublisher

# Resources

- Powerful access to low-level resources
- Avoids direct use of classloaders
- Simplifies exception handling
- Several built-in implementations:
  - ClassPathResource
  - FileSystemResource
  - URLResource

```
public interface Resource
    extends InputStreamSource
{
    boolean exists();
    boolean isOpen();
    URL getURL();
    File getFile();
    Resource createRelative( String p );
    String getFileName();
    String getDescription();
}

public interface InputStreamSource()
{
    InputStream getInputStream();
}
```

# Factory beans

- Bean that produces objects
  - Defined as normal bean but returns the produced object
  - Must implement the FactoryBean interface

```
public class DatabaseConfigFactoryBean implements FactoryBean<DatabaseConfig>
{
   public DatabaseConfig getObject() throws Exception
   {
      // Create and return DatabaseConfig object
   }

   // Must also implement getObjectType() and isSingleton()
}
```

```
public class StudentDao
{
   private DatabaseConfig databaseConfig;

   // set-method
}
```

# Convenient ApplicationContext instantiation for web apps

- In web.xml

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

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

# Summary

- IoC Container
- Bean scopes
- Bean lifecycle customization
- Internationalization
- Appliction Events / Listeners
- Resources
- FactoryBeans

- Spring reference documentation
- http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/index.html