Spring Framework
Module 2 – Components Model (IoC, DI)

Evgeniy Krivosheev
Andrey Stukalenko
Vyacheslav Yakovenko
Last update: Feb, 2012

Contents

- Inversion of Control (IoC) / Dependency Injection (DI)
- Spring Framework's IoC Containers
- Working with IoC Container
- Beans as Components
- External Dependencies
- Dependency Injection (DI)
- Autowiring
- Annotation-based configuration
- Bean Scopes
- Bean Lifecycle
- Additional Features of ApplicationContext

Spring Framework :: IoC / DI

- Inversion of Control (IoC) pattern is the basis for Spring
  - “Hollywood Principle” – Don't call me, I'll call you (Could you recall any design pattern from GoF catalog to which this slogan can be applied?);
  - Basic idea is to eliminate dependency of application components from certain implementation and to delegate IoC container rights to control classes instantiation;
- Martin Fowler suggested the name Dependency Injection (DI) because it better reflects the essence of the pattern (http://martinfowler.com/articles/injection.htm)

Spring Framework :: IoC / DI

- **Task:**
  - Two components A and B
  - Component A uses functionality of Component B

- **Classical "direct" solution:**

```
public class A {
    private B m_b = new B();
    public doSomething() {
        b.do();
    }
}
```

- **Problems:**
  - Class A directly depends on Class B;
  - You can't test A in isolation from B (if basis for B is needed than it is needed for testing A as well);
  - Lifecycle of B object is controlled by A, so you can't use the same object in other places;
  - You can't "substitute" B for different implementation;

Spring Framework :: IoC / DI

- **Implementing interface:**

```
public class A {
   private IB m_ib = new B();
   public void doSomething() {
      m_ib.do();
   }
}
public class B implements IB { … }
public interface IB { … }
```

- **Problems:**
  - Class A directly depends on Class B;
  - You can't test A in isolation from B (if basis for B is needed than it is needed for testing A as well);
  - Lifecycle of B object is controlled by A, so you can't use the same object in other places;
  - You can only substitute B for different implementation by changing code;
  - Outcome: problems have not been solved.

Spring Framework :: IoC / DI

- **Service Locator:** there is a factory that returns required interface implementation by identifier:

```
public class A {
    private IB m_ib = serviceLocator.getService(IB.class);
    public void doSomething() {
        m_ib.do();
    }
}
```

- **Advantages:**
  - Lifecycle of B is controlled by locator;
  - Class A doesn't depend on IB implementation, thus you can use any implementation you need;
- **Problems:**
  - Class A depends on Service Locator;
  - You can test A in isolation from B, but it is likely that for that you'll have to configure locator uncommonly;

Spring Framework :: IoC / DI

- **Dependency Injection** – component A doesn't lookup dependencies, but get them from external modules (for instance, via constructor or setter method):

```
public interface IB { … }
public class A {
   private IB m_ib;
   public A(IB value) { m_ib = value; }
   public void doSomething() {
      m_ib.do();
   }
}
```

- **Advantages:**
  - A doesn't depend on anything;
  - Testing A is much easier;
- **Problems:**
  - The problem is not solved on a large scale;
  - … = new A(new B(new C(new D(new E(…);
  - Serious problems related to objects lifecycle control appear;

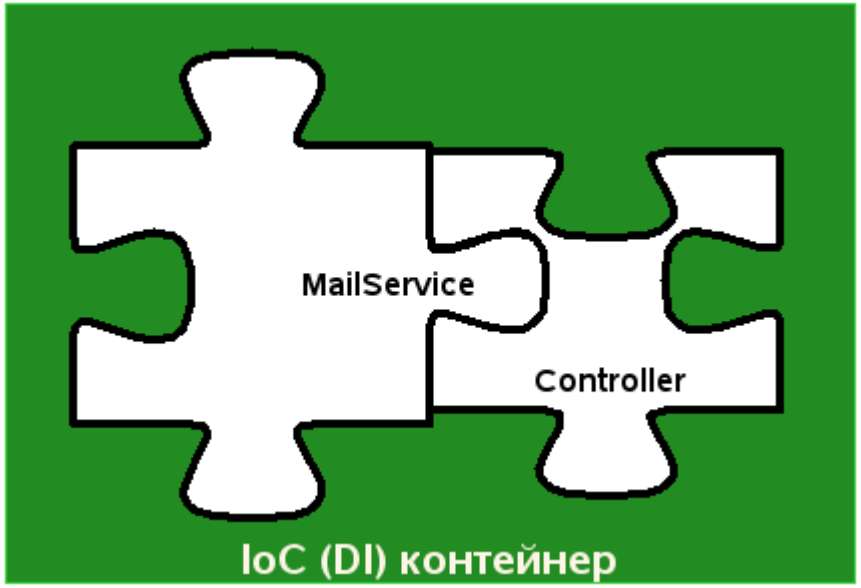
Spring Framework :: IoC / DI

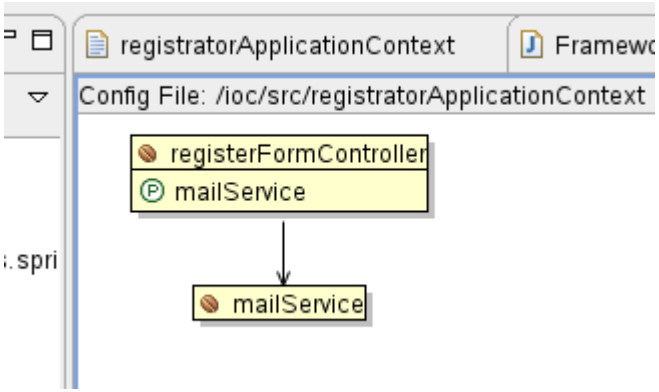
- **Solution is Inversion of Control container**

```
// resolve all dependencies
A aobj = container.resolve(A.class);
```

- It is similar to Service Locator;
- If requesting an object of any type, the container decides which one should be returned;
- There is a dependency diagram for each type recorded in container (it describes what should be sent to constructor, what should be given to various properties and etc.);
- Contains associations: which object should be returned to the requested identifier;
- When creating request object, container creates a relevant object (or matches ready one) for every dependency. Dependencies of this object are solved recursively as well;
- Controls lifecycle of created objects;
- Potentially, IoC container can be used as Service Locator, but it will be better if you avoid it.

Spring Framework :: IoC / DI





Spring Framework :: IoC / DI

Advantages of IoC containers:

- Dependency management and change without recompiling;
- Facilitates reusing classes or components;
- Simplified unit testing;
- More pure code (classes don’t initiate auxiliary objects);
- Only interfaces with implementations that are subject to change in current or future projects should be loaded in IoC container.

Spring Framework :: IoC Containers

LUXOFT TRAINING

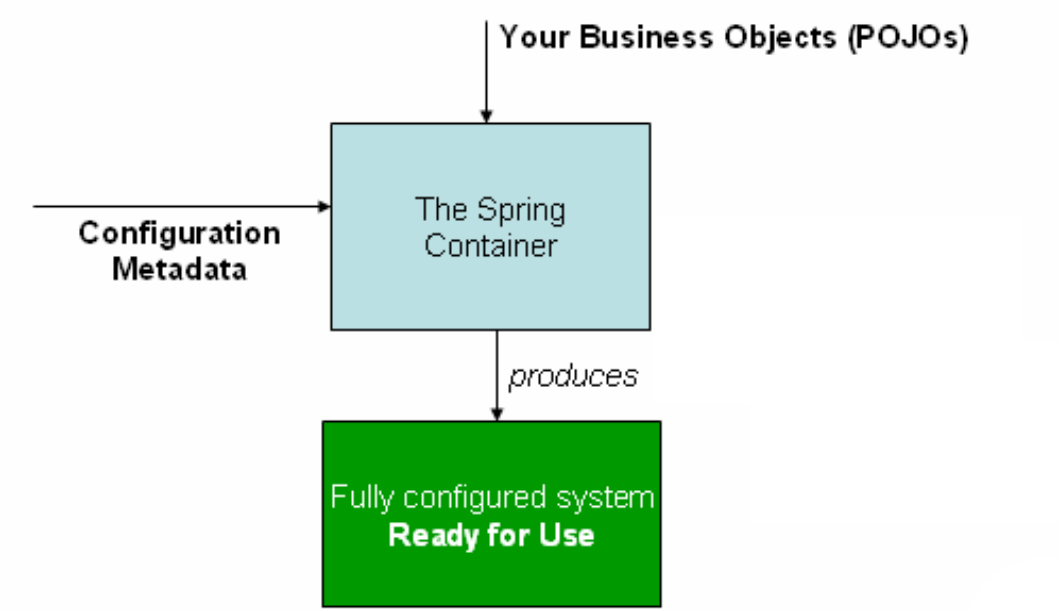
- BeanFactory is a central IoC container interface in Spring Framework (implementation used is **XmlBeanFactory**):
  - BeanFactory provides only basic low-level functionality;
- ApplicationContext is an interface enhancing BeanFactory and adding performance to basic container features:
  - Simple integration with Spring AOP;
  - Work with resources and messages;
  - Event handling;
  - Specific application contexts (for instance, WebApplicationContext);
- It is Application Contexts that are used in a real life;
- BeanFactory could be used in exceptional cases, for example, if integrating Spring with a framework or when resources are critical and only IoC container is required.

Spring Framework :: IoC Containers

- There are several available implementations of ApplicationContext. The main are:
  - `GenericXmlApplicationContext` (since v.3.0);
  - `ClassPathXmlApplicationContext`;
  - `FileSystemXmlApplicationContext`;
  - `WebApplicationContext`;
- XML is a traditional way to configure container, though there are various ways of configuring metadata (annotations, Java code, etc.);
- In most cases it is easier and faster to use annotation-based configuration. However, you have to remember that annotation-based configuration has some restrictions and introduces additional code-level dependencies;
- Generally, user (developer) won't have to initiate Spring IoC container on his or her own;

Spring Framework :: Working with IoC Container

In general, work of Spring IoC container can be represented as follows:



- When instantiating and initiating container, your application classes are combined with metadata (container configuration) and at the output you get fully configured and ready-to-work application.

Spring Framework :: Working with IoC Container

- Instantiating a container

```
ApplicationContext ctx =
  new ClassPathXmlApplicationContext("services.xml");
```

Spring Framework :: Working with IoC Container

LUXOFT TRAINING

- Configuration example:

```
<?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-3.0.xsd">

  <bean id="myService" class="foo.bar.ServiceImpl">
   <property name="param1" value="some value" />
   <property name="otherBean" ref="otherBeanService" />
  </bean>

  <bean id="otherBeanService" class="..."/>

  <!-- more bean definitions go here -->
</beans>
```

Spring Framework :: Beans as components

LUXOFT TRAINING

- In Spring any Java objects managed by container are referred to as *beans*;
- Bean is an ordinary Java object (POJO);
- Having default (no-argument) constructor is not necessary;

- When configuring you may define:
  - The unique identifier for a bean (ID);
  - Fully qualified class name;
  - Bean behavior in the container (scope, lifecycle callbacks, etc.);
  - Dependencies from other beans;
  - Other configuration settings (for example, connection pool size).

Spring Framework :: Beans as components

- Instantiation with constructor:

```
<bean id="example1"
    class="ru.luxoft.training.samples.Example" />
```

- Instantiation using a static factory method:

```
<bean id="clientService"
    class="ru.luxoft.training.samples.ClientService"
    factory-method="createInstance" />
```

- Instantiation using an instance factory method

```
<bean id="serviceFactory"
    class="examples.DefaultServiceFactory" />

<bean id="clientService"
    factory-bean="serviceFactory"
    factory-method="createClientServiceInstance" />
```

Spring Framework :: lazy-initialization mode

- For a specific bean:

```
<bean id="lazy" class="…" lazy-init="true" />
```

- For all beans in container:

```
<beans default-lazy-init="true">
…
</beans>
```

- If singleton bean depends on lazy bean, then lazy bean is instantiated with instantiating singleton bean.

Spring Framework :: External Dependencies

- **The only mandatory external dependency in Spring is Jakarta Commons Logging API (JCL)**
- Resolving dependencies at runtime
- Alternative to JCL is SLF4J:
  - Exclude commons.logging from dependencies and classpath
  - Add SLF4J-JCL on the classpath (jcl-over-slf4j)
  - Add SLF4J-API on the classpath
- To use Log4J you have to:
  - Put log4j library on the classpath
  - Put log4j configuration file (log4j.properties или log4j.xml) in the root of the classpath

Exercises

- №3: “Hello, World” example for Spring Framework:
  - 20 min for practice;
  - 10 min for discussion;

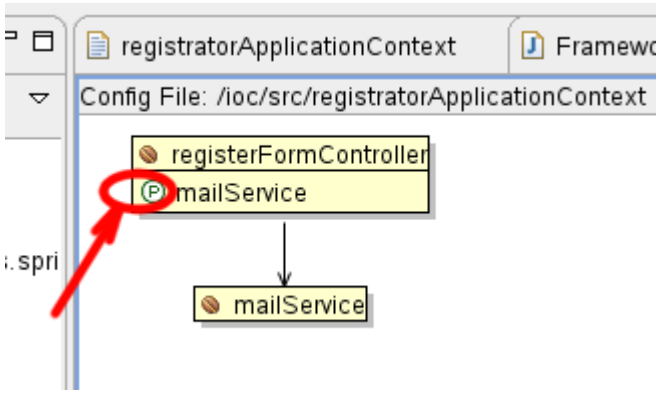
Spring Framework :: Aliasing

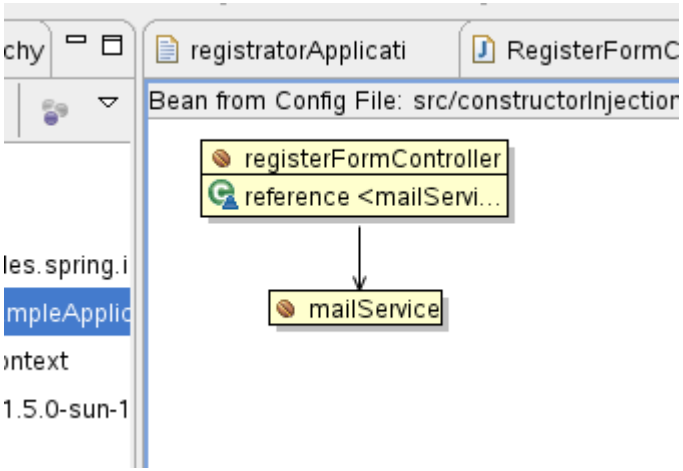
```
<alias fromName="originalName" toName="aliasName" />
```

- In this case bean named originalName may be referred to as aliasName;
- Generally, you often use this when application architecture implies possible extensions, but at the moment there is no need for specific sections (thus, there is no meaning in propagating additional objects).

Spring Framework :: DI

- There are two main types of Dependency Injection (DI):
  - Constructor DI
  - Setter DI





Spring Framework :: Constructor DI

```
public class ConstructorInjection {
   private Dependency dep;
   private String descr;

   public ConstructorInjection(Dependency dep, String descr) {
      this.dep = dep;
      this.descr = descr;
   }
}

<bean id="dependency" class="Dependency" />
<bean id="constrInj" class="ConstructorInjection">
   <constructor-arg ref="dependency" />
   <constructor-arg value="Constructor DI" />
</bean>
```

Spring Framework :: Constructor DI

- Circular dependency:

```
class A {
    private B b;
    A(B b) {
        this.b = b;
    }
}
```

```
class B {
    private A a;
    B(A a) {
        this.a = a;
    }
}
```

- If configuring classes with Constructor DI, *BeanCurrentlyInCreationException* will be thrown

- Solution is to substitute Constructor DI for Setter DI in one or both classes

Spring Framework :: Setter DI

```
public class SetterInjection {
   private Dependency dep;
   private String descr;

   public void setDep(Dependency dep) {
      this.dep = dep;
   }
   public void setDescr(String descr) {
      this.descr = descr;
   }
}

<bean id="dependency" class="Dependency" />
<bean id="setterInj" class="SetterInjection">
   <property name="dep" ref="dependency" />
   <property name="descr" value="Setter DI" />
</bean>
```

Spring Framework :: Autowiring

LUXOFT TRAINING

- Spring is able to autowire (add dependencies) beans instead of <ref>;
- In some cases it can significantly reduce the volume of configuration required;
- Can cause configuration to keep itself up to date as your object model evolve (for example, if you need to add an additional dependency, that dependency can be satisfied automatically);
- Autowiring by type can only work if there is exactly one bean of a property type;
- Harder for reading and checking dependencies than explicit wiring;

- Specified with autowire attribute in bean definition

```
<bean id="" class="" autowire="value" />
```

Spring Framework :: Autowiring

LUXOFT TRAINING

- Autowiring modes:
  - **no**: no autowiring at all. This is the default;
  - **byName**: autowiring by property name. This option will inspect the container and look for a bean with ID exactly the same as the property which needs to be autowired. If such a bean cannot be found, the object is not autowired;
  - **byType**: autowiring by type. Works only if there is exactly one bean of property type in container. If there is more than one, then **UnsatisfiedDependencyException** is thrown;
  - **constructor**: container looks for a bean (or beans) of the constructor argument type. If there is more than one bean type or more than one, then **UnsatisfiedDependencyException** is thrown;

Spring Framework :: Annotation-based Configuration

- Spring container may be configured with the help of annotations;

- Basic supported annotations:
  - @Required
  - @Autowired
  - @Component

- For annotation-based configuration you should indicate in configuration of Spring container the following:

```
<context:annotation-config/>
```

Spring Framework :: Annotation-based Configuration

LUXOFT TRAINING

@Required

- Applies to bean property setter method;
- Indicates that the affected bean property must be populated at configuration time (either through configuration or through autowiring);
- If the affected bean property has not been populated the container will throw an exception. This allows to avoid unexpected NullPointerException in system operation;

```
public class SimpleMovieLister {
   private MovieFinder movieFinder;

  @Required
  public void setMovieFinder(MovieFinder movieFinder) {
      this.movieFinder = movieFinder;
  }
}
```

Spring Framework :: Annotation-based Configuration

LUXOFT TRAINING

@Autowired

- Applied to:
  - Setter methods;
  - Constructors;
  - Methods with multiple arguments;
  - Properties (including private ones);
  - Arrays and typed collections (ALL beans of relevant class are autowired)

- Can be used with @Qualifier(“name”).If this is the case, a bean with relevant ID is autowired;

- By default, if there is no matching bean, an exception is thrown. This behavior can be changed with @Autowired(required=false);

Spring Framework :: Annotation-based Configuration

LUXOFT TRAINING

@Component

- Used for specifying Spring components without XML configuration
- Applies to classes

- Serves as a generic stereotype for every Spring-managed component
- It is recommended to use more specific stereotypes*:
  - **@Service**
  - **@Repository**
  - **@Controller**
- Generally, when you doubt which stereotype shall be used, use @Service
- To automatically register beans through annotations, specify the following command in container configuration:

```
<context:component-scan base-package="org.example"/>
```

* - history of stereotypes and their purpose traces back to Eric Evans' book *Domain-Driven Design: Tackling Complexity in the Heart of Software*

Spring Framework :: Bean Scopes

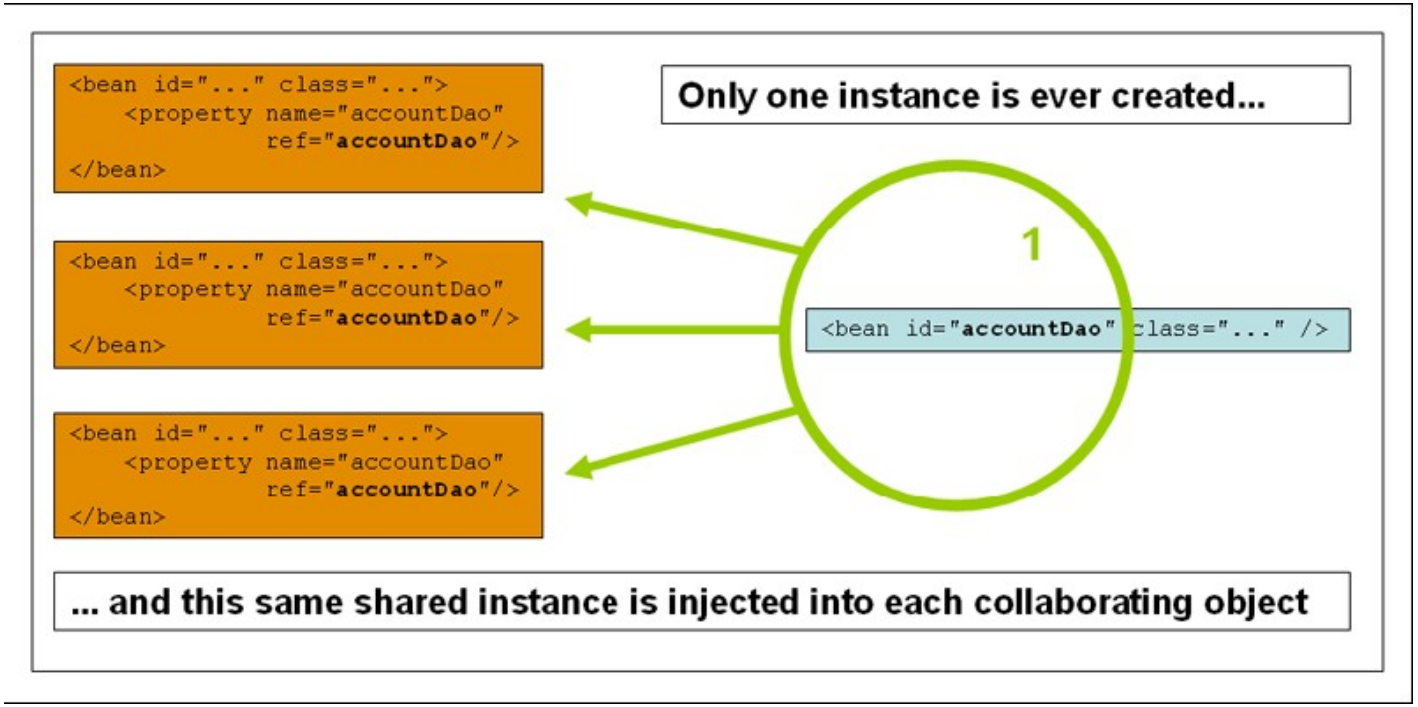
LUXOFT TRAINING

Bean Scopes

- General: Bean Scopes
  - Singleton
  - Prototype
- Web-specific:
  - Request
  - Session
  - Global session

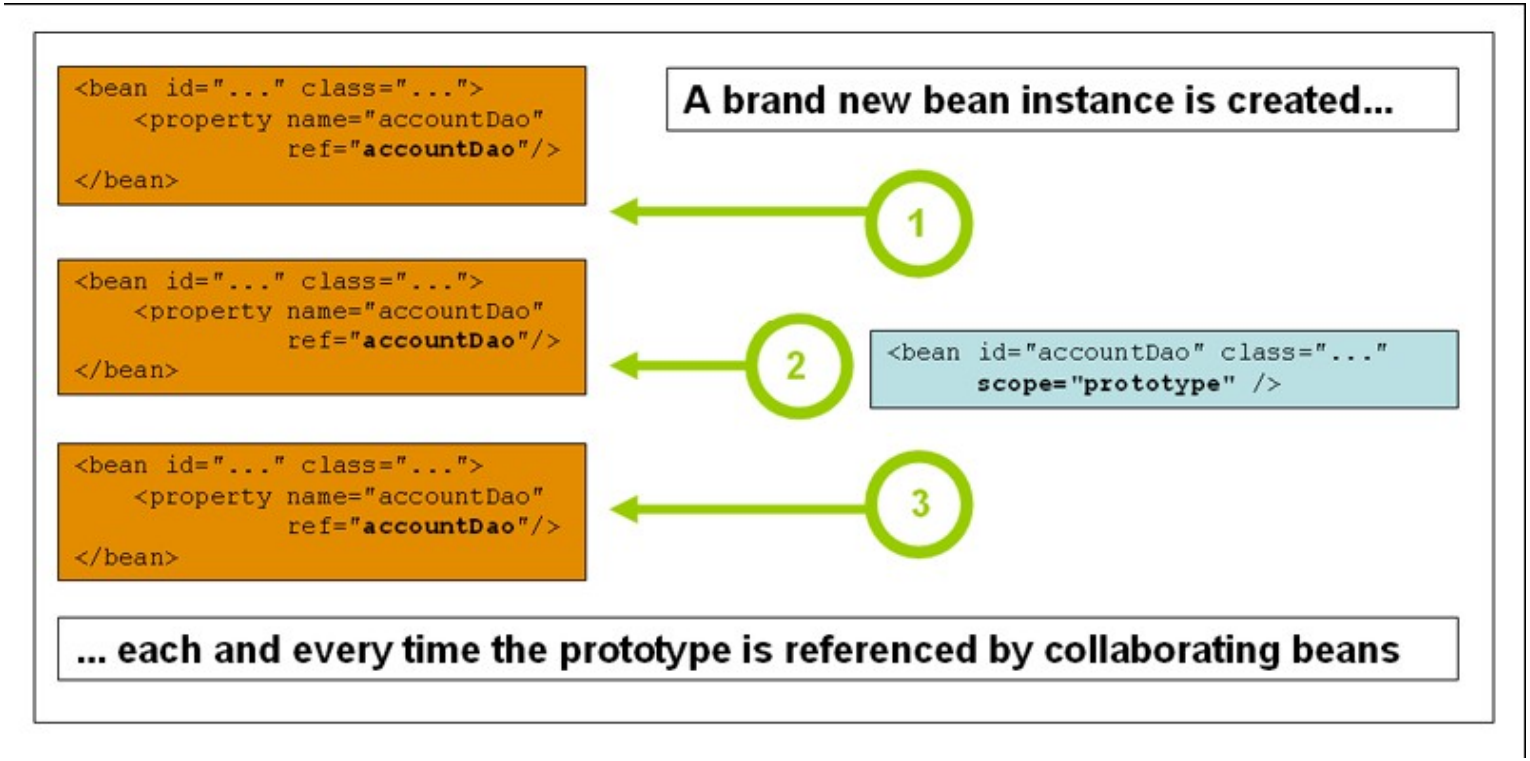
Spring Framework :: Bean Scopes

- **Singleton**
  - By default
  - Single bean instance in container

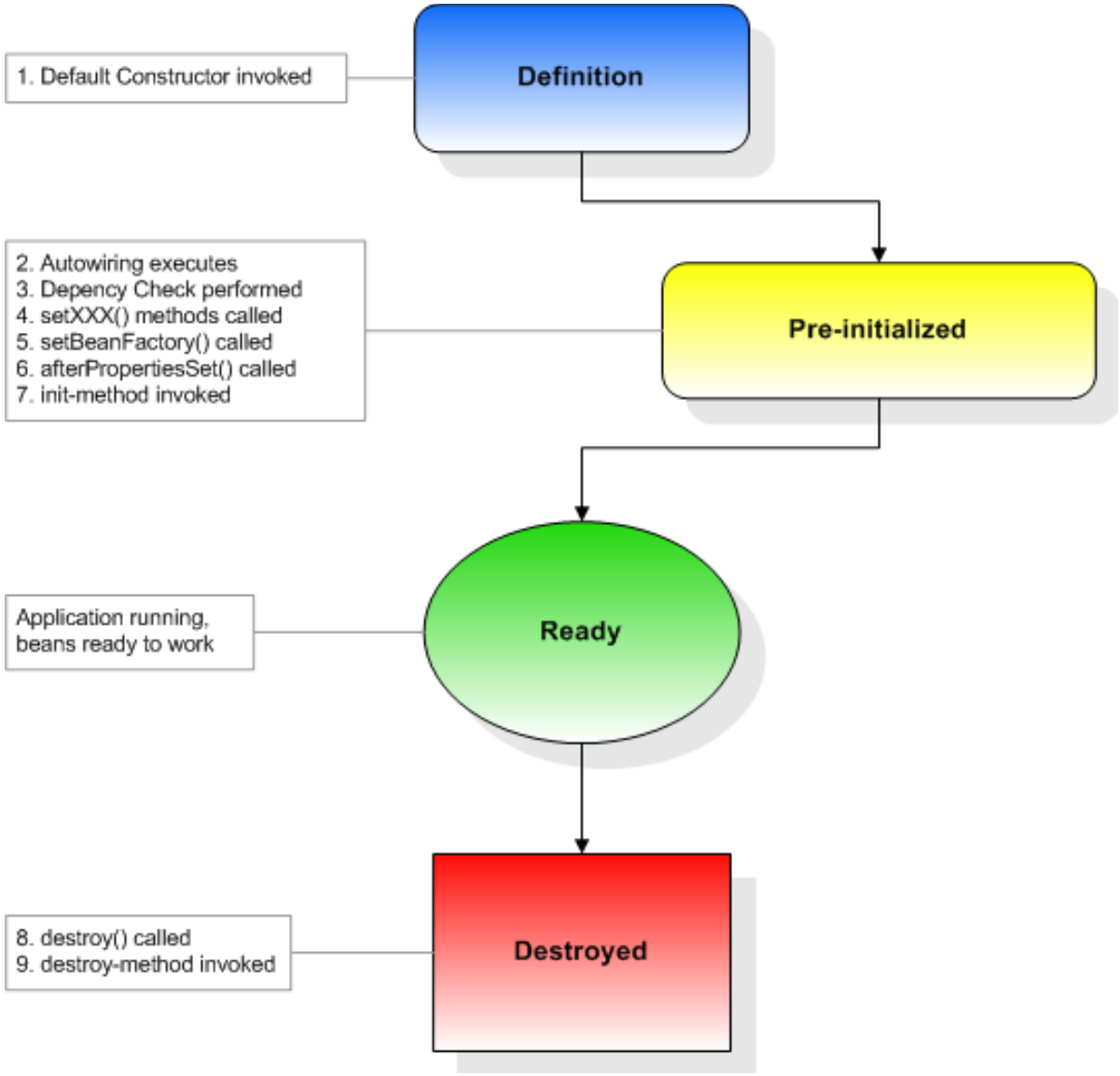


Spring Framework :: Bean Scopes

- **Prototype**
  - A brand new bean instance is created every time it is injected into another bean or it is requested via getBean().



Spring Framework :: Bean Lifecycle



Spring Framework :: Bean Lifecycle

LUXOFT TRAINING

- BeanFactory or ApplicationContext is created;
- Each bean has a dependency in terms of properties or constructor arguments;
- Each property or constructor argument is defined either by precise value or by link to other bean;
- It shall be possible to convert property/constructor argument values from initial format to format required (see PropertyEditor);
- Container validates configuration of every single bean;
- Properties are initialized at the moment when a bean is actually instantiated;
- Singleton beans are instantiated simultaneously with constructor (if not lazy);
- The rest only when they are requested.

Spring Framework :: Bean Lifecycle

Managing bean by implementing Spring interfaces

- Creating
  - Implement interface InitializingBean
  - Override method afterPropertiesSet()
- Deleting
  - Implement interface DisposableBean
  - Override method destroy()

Spring Framework :: Bean Lifecycle

Managing bean via code without dependence on Spring:

- Add methods for initialization and/or deletion in a specific bean and indicate them in bean declaration:

```
<bean id="example" class="Example"
  init-method="init"
  destroy-method="cleanup" />
```

- Methods for creating and/or deleting can be defined for all beans inside the container:

```
<beans default-init-method="init"
  default-destroy-method="cleanup">
```

Spring Framework :: Additional Features of ApplicationContext

- To access context (for example, for event publishing) a bean has to only implement interface ApplicationContextAware

```
public class CommandManager implements ApplicationContextAware {

   private ApplicationContext applicationContext;

   public void setApplicationContext(
         ApplicationContext applicationContext)
         throws BeansException
   {
      this.applicationContext = applicationContext;
   }
}
```

Spring Framework :: Additional Features of ApplicationContext

LUXOFT TRAINING

- Event handling inside ApplicationContext is provided through
  - ApplicationEvent class
  - ApplicationListener interface

- Every time an event sets in, all beans implementing ApplicationListener interface that are registered in container are notified.

- ApplicationEvent : basic implementations:
  - **ContextRefreshedEvent** is creation or update of ApplicationContext:
    - Singletons have been created
    - ApplicationContext is ready for use
  - **ContextClosedEvent**
    - After using close() method
  - **RequestHandledEvent**
    - For web applications only

Spring Framework :: Additional Features of ApplicationContext

- Implementing standard events:

```
public class MyBean implements ApplicationListener {
   Public void onApplicationEvent(ApplicationEvent event) {
      …
   }
}
```

- Publishing custom events*:

```
public class CustomEvent extends ApplicationEvent {
   public CustomEvent (Object obj) {
      super(obj);
   }
}
context.publishEvent(new CustomEvent(new Object()));
```

* - As of Spring 3.0, an ApplicationListener can generically declare the event type that it is interested in. When registered with a Spring ApplicationContext, events will be filtered accordingly, with the listener getting invoked for matching event objects only. Interface ApplicationListener<E extends ApplicationEvent>

Spring Framework :: Additional Features of ApplicationContext

LUXOFT TRAINING

- The ApplicationContext interface extends an interface called MessageSource, and therefore provides internationalization (i18n) functionality
- When it is loaded it automatically searches for a MessageSource bean in configuration (bean shall be inherited from MessageSource and have id="messageSource")
- If the ApplicationContext cannot find the bean in the context, an empty DelegatingMessageSource is instantiated in order to guarantee proper handling of relevant methods

Spring Framework :: Additional Features of ApplicationContext

LUXOFT TRAINING

Example of Message Source configuration:

```
<bean id="messageSource"
  class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>format</value>
      <value>exceptions</value>
      <value>windows</value>
    </list>
  </property>
</bean>
```

- Contents of format_**en_GB**.properties

```
message=Alligators rock!
```

- Contents of format_**ru_RU**.properties

```
message=Крокодилы – нереально круты!
```

Spring Framework :: Collections Initialization

```
<bean id="..." class="...">
<property name="someList">
  <list>
      <value>a list element followed by a reference</value>
      <ref bean="myDataSource" />
  </list>
</property>
<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
  <map>
      <entry key="an entry" value="just some string"/>
      <entry key ="a ref" value-ref="myDataSource"/>
  </map>
</property>
<!-- results in a setSomeSet(java.util.Set) call -->
<property name="someSet">
  <set>
      <value>just some string</value>
      <ref bean="myDataSource" />
  </set>
</property>
</bean>
```

Exercises

- №4: Developing primitive application:
  - 50 min for practice;
  - 10 min for discussion;

LUXOFT TRAINING

Any questions!?