Spring Framework
Inversion of control

Part 1

Spring Framework - Introduction

- Spring is a **lightweight** yet at the same time **flexible** and **universal** framework used for creating Java SE and Java EE applications
- Spring is a framework with an **open source code**
- Spring is an **application framework**, not a layer framework
- Spring includes several separate frameworks

Spring Framework - Introduction

- **Rod Johnson** created Spring in **2003**
- Spring took its rise from books ***Expert One-on-One Java J2EE Design and Development*** and ***J2EE Development Without EJB***
- The basic idea behind Spring is to **simplify** the traditional approach to designing J2EE applications

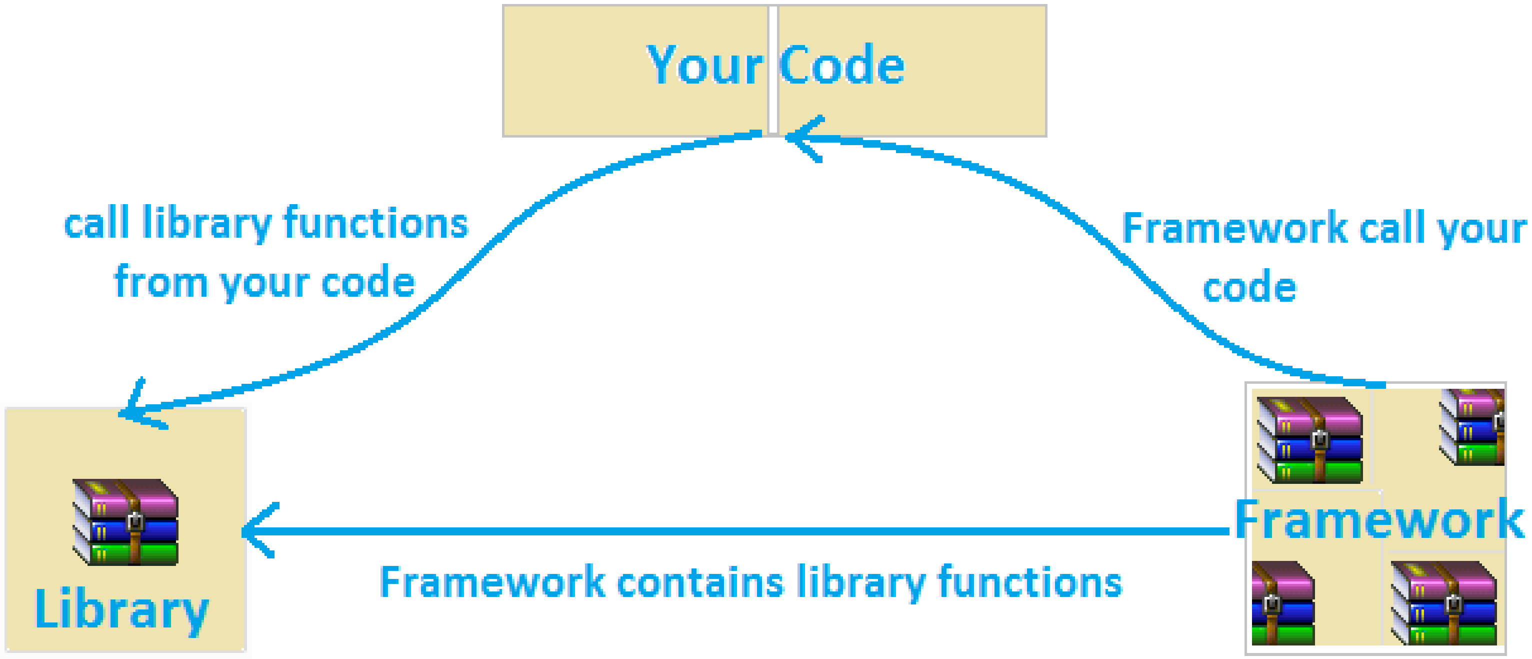
Rod Johnson

Rod Johnson is an enterprise Java architect specializing in scalable web applications. He has worked with both Java and J2EE since their release, and he is a member of JSR 154 Expert Group defining the Servlet 2.4 specification.

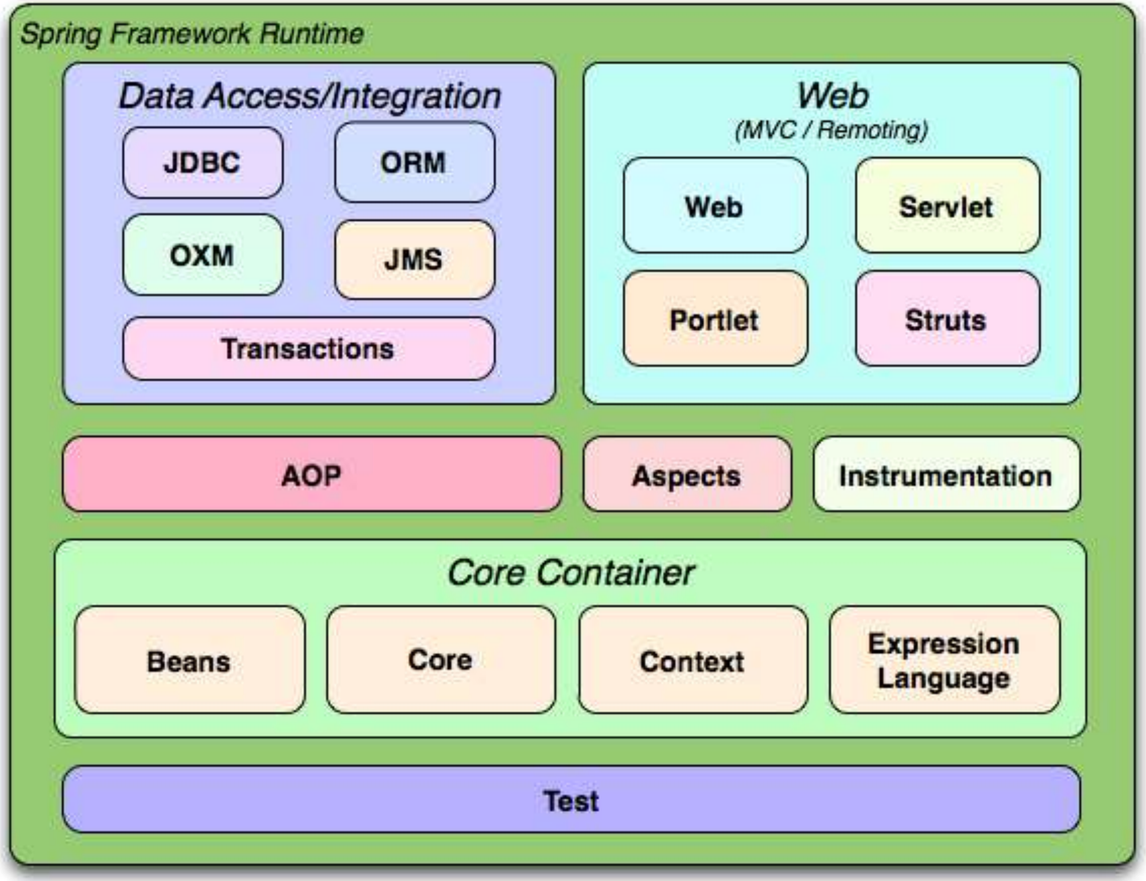
expert one-on-one

J2EE™ Design and Development

Differences between Library and Framework


Your Code
call library functions from your code
Framework call your code
Library
Framework
Framework contains library functions

Spring Framework - Core Container



Core Container consists of:

- spring-beans
- spring-core
- spring-context
- spring-context-support
- spring-expression

Spring Framework - Framework structure

- **Spring Framework** - **Java platform** that provides comprehensive infrastructure for developing Java applications
- Handles the infrastructure so you can focus on your application
- Enables you to build applications from **"plain old Java objects" (POJOs)** and to apply enterprise services to **POJOs**

Spring Framework – Object dependencies

- Spring implements various design patterns:

  - **Factory**
  - **Builder**
  - **Proxy**

- The Spring Framework **Inversion of Control (IoC)** provides a means of composing disparate components into a fully working application.

Spring Framework – Object Dependencies

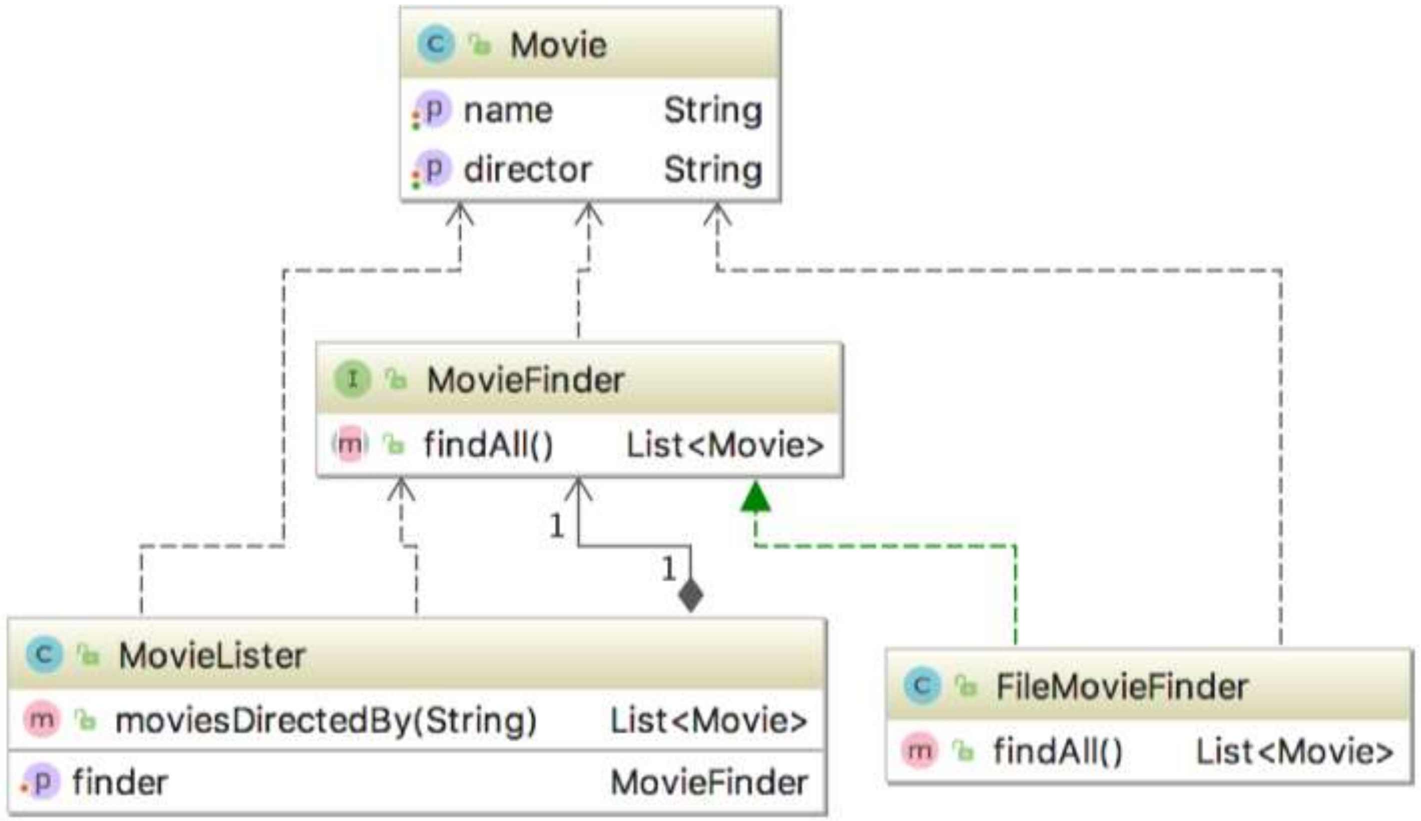
```java
public class DirectMovieLister
{
    private FileMovieFinder finder;

    public DirectMovieLister()
    {
        finder = new FileMovieFinder();
    }

    public List<Movie> moviesDirectedBy(String director)
    {
        List<Movie> result = new ArrayList<>();

        for (Movie movie : finder.findAll())
        {
            if (movie.getDirector().equals(director))
            {
                result.add(movie);
            }
        }
        return result;
    }
}
```

Spring Framework – Object Dependencies

What if:

- You have a completely different form of storing a movie listing: XML file, SLQ database, http service, etc?
- I need to test **DirectMovieLister** separately from **FileMovieFinder**?

Spring Framework – Object Dependencies



Spring Framework – Object Dependencies

In this case we can:

- Use different implementations of **FileMovieFinder** in different deployments
- Change implementations at runtime
- Use single instance of **FileMovieFinder** for several instances of **MovieLister**

Spring Framework – Object Dependencies

So the core problem is:

How do we assemble these services into an application?

Spring Framework – Object dependencies

We have to automate this:

```
public static void main(String[] args)
{
      MovieFinder finder = new FileMovieFinder();

      MovieLister lister = new MovieLister();

      lister.setFinder(finder);

      List<Movie> filtered = lister.moviesDirectedBy("Spielberg");
}
```

And we need a Dependency Injection Framework to do it for us!

Spring Framework – Object dependencies

1. Object **defines** dependencies via configuration
2. The **container then injects** those dependencies **when it creates** a specific object

Spring Framework – Object Dependencies

Inversion of Control Container approach

application-context.xml

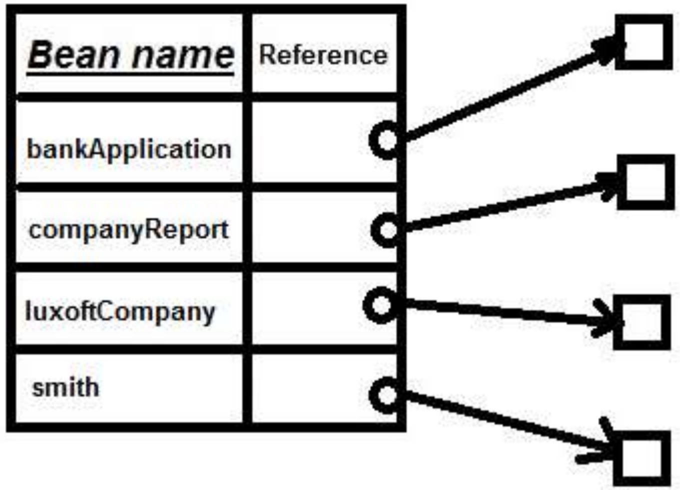
```
<bean id="fileMovieFinder" class="com.luxoft.springioc.movies.FileMovieFinder">
     <property name="fileName" value="movies.txt" />
</bean>

<bean id="movieLister" class="com.luxoft.springioc.movies.MovieLister">
     <property name="finder" ref="fileMovieFinder" />
</bean>
```

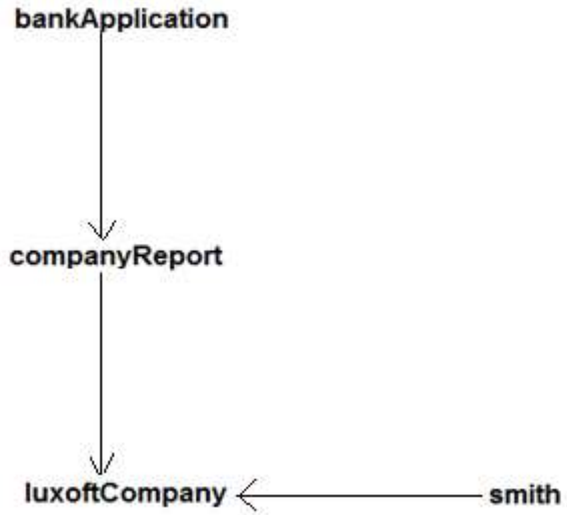
```
public static void main(String[] args)
{
     ClassPathXmlApplicationContext context =
           new ClassPathXmlApplicationContext("movies/application-context.xml");

      MovieLister lister = (MovieLister) context.getBean("movieLister");
      List<Movie> filtered = lister.moviesDirectedBy("Spielberg");
}
```

ex. movies

Internal Structure of Application Context

- The application context internally keeps a map to provide access to the managed objects. The creation of the objects and their relationship is managed by the container through IoC/DI.





Spring Framework – Object Dependencies

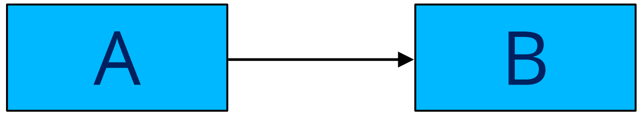
Inversion of Control Container approach

Advantages:

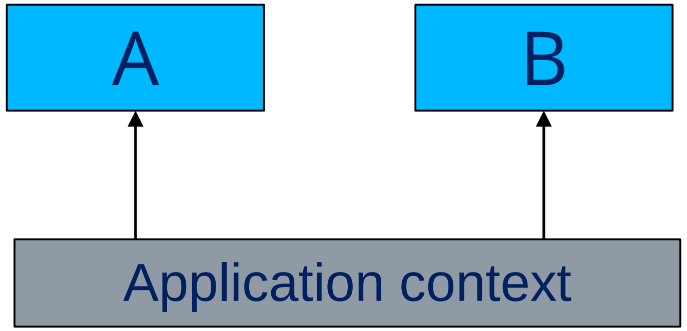
- The container creates the necessary objects and manages their lifetime
- **MovieLister** and **FileMovieFinder** are independent objects
- Application Context creates beans and injects configured dependencies
- It’s very easy to make changes to object dependencies in the system

Spring Framework – Object Dependencies

Traditional approach: dependencies inside the code



IoC: objects know nothing about each other



- *Creates B and initializes it*
- *Creates A and initializes it*
- *Injects B to A*

Spring Framework – IoC / DI

Inversion of Control (IoC) or Dependency Injection (DI) pattern is the base for Spring.

- The “Hollywood Principle” – Don't call me, I'll call you

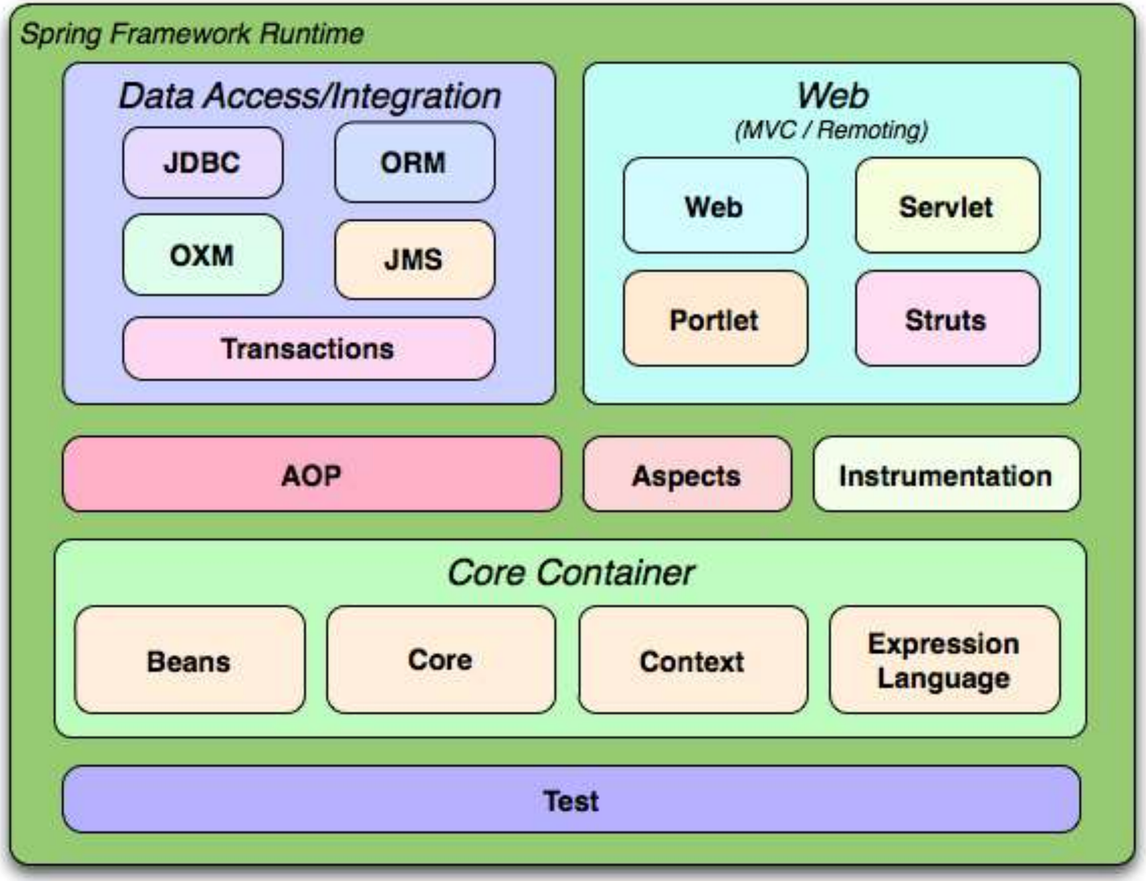
- The basic idea is to eliminate the dependency of application components from certain implementation and to delegate DI container rights to control classes instantiation and initialization

Spring Framework – IoC / DI

Advantages of IoC containers:

- Dependency management and applying changes without recompiling
- Facilitates reusing classes or components
- Simplified unit testing
- Cleaner code (classes do not initiate auxiliary objects)
- It is especially recommended to insert the objects for which the implementation may change to the DI container

Spring Framework - Core Container



Core Container consists of:

- spring-beans
- spring-core
- spring-context
- spring-context-support
- spring-expression

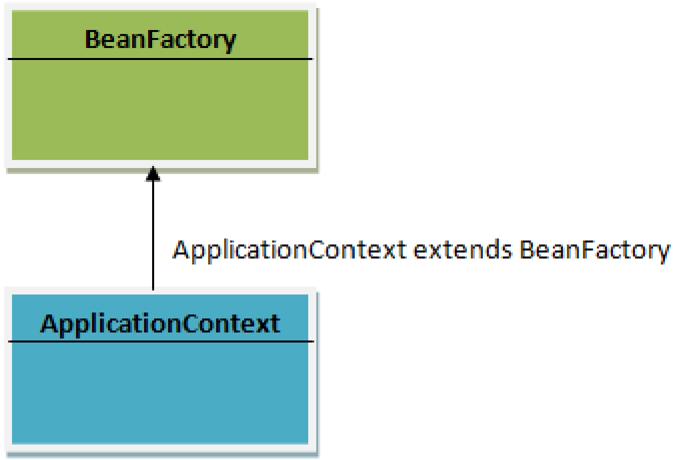
Spring Framework – IoC containers

- The ***BeanFactory*** is a central **IoC** container interface into the **Spring Framework**
- Implementation of the factory pattern
- Most common implementation: ***XmlBeanFactory***
- ***BeanFactory*** provides only basic low-level functionality

Spring Framework – IoC Containers

- **`ApplicationContext extends BeanFactory`** and adds:
  - Event handling
  - Internationalization
  - Work with resources and messages
  - Simple integration with Spring AOP
  - Specific application contexts

  (for example, ***ClassPathXmlApplicationContext***)



Spring Framework – IoC Containers

- The **ApplicationContext** interface is the focal point of the **Context** module
- **ApplicationContext**s are used in real life
- **BeanFactory** could be used in exceptional cases:
    - Integrating Spring with a framework (backward compatibility is necessary)
    - Resources are critical and only IoC container is required

Spring Framework – IoC containers

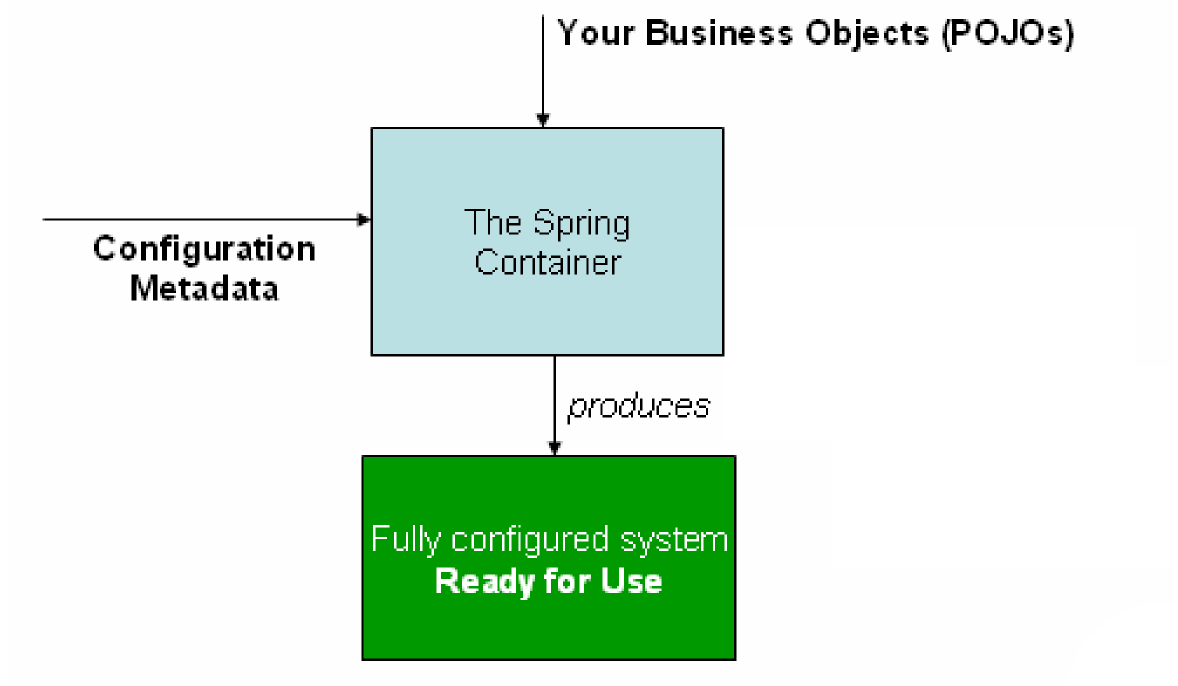
- Most widely used implementations of **ApplicationContext**:
  - **GenericXmlApplicationContext** (since v.3.0)
  - **ClassPathXmlApplicationContext**
  - **FileSystemXmlApplicationContext**
- XML is a traditional way to configure a container
- It is easier and faster to use annotation-based configuration, but this one has some restrictions and introduces additional code-level dependencies

Spring Framework – IoC Containers

```
ApplicationContext context =
   new GenericXmlApplicationContext("classpath:context.xml");

ApplicationContext context =
   new ClassPathXmlApplicationContext("context.xml");


ApplicationContext context =
   new GenericXmlApplicationContext("context.xml");

ApplicationContext context =
   new FileSystemXmlApplicationContext("context.xml");
```

Spring Framework – IoC Containers

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



Spring Framework – Maven Configuration

```xml
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.1.3.RELEASE</version>
</dependency>
```

Spring Framework – Working with IoC Container

Container creation:

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("application-context.xml");

MovieLister lister = (MovieLister) context.getBean(“movieLister");

Ex1TestBean testBean = context.getBean(Ex1TestBean.class);
```

ex.1

Spring Framework – Working with IoC Container

```
ApplicationContext context = new ClassPathXmlApplicationContext(
     new String[]{"example02/services.xml", "example02/daos.xml"});

Service service = context.getBean(ServiceBean.class);
service.printNames();
```

services.xml

```
<bean id="service" class="com.luxoft.springioc.example02.ServiceBean" >
     <property name="dao" ref="dao" />
</bean>
```

daos.xml

```
<bean id="dao" class="com.luxoft.springioc.example02.DaoBean"/>
```

ex. 2

Spring Framework – Bean Creation

no-args constructor:

```
<bean id="clientService" class="com.luxoft.springioc.ClientService"/>
```

ex.3

Spring Framework – Bean Creation

Factory method:

```
<bean id="clientService" class="com.luxoft.springioc.ClientService"
                        factory-method="createInstance" >
    < constructor-arg value="Software Development" />
</bean>

public static ClientService createInstance(String serviceType ) {
    ClientService clientService = new ClientService();
    clientService.setServiceType(serviceType);
    if (serviceType.equals("Software Development")) {
        clientService.setRemote(true);
    }
    // possibly perform some other operations
    // with clientService instance
    return clientService;
}
```

ex. 4

Task for ex. 4

Create `BusinessService` which will be retrieved by a factory method. A `BusinessService` is defined by company name and by domain. If company name is `"Luxoft"`, domain will be `"IT"`. Otherwise, domain will be `"Financial"`.

Spring Framework – Bean Creation

Factory class:

```
<bean id="serviceFactory" class="com.luxoft.springioc.DefaultServiceFactory"/>
<bean id="clientService" factory-bean="serviceFactory"
    factory-method="createClientServiceInstance" >
        <constructor-arg value="Retailing" />
</bean>

public ClientService createClientServiceInstance(String serviceType) {
    ClientService clientService = new ClientService();
    clientService.setServiceType(serviceType);
    if (serviceType.equals("Software Development")) {
        clientService.setRemote(true);
    }
    return clientService;
}
```

ex.5

Task for ex. 5

- Use BusinessService to be retrieved by DefaultServiceFactory

Spring Framework – Lazy Initialization

Lazy initialization is used to postpone bean creation until it is first addressed.

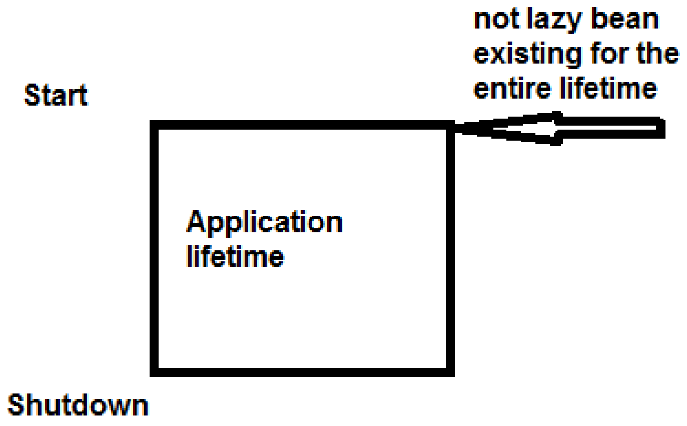
Lazy initialization of single bean:

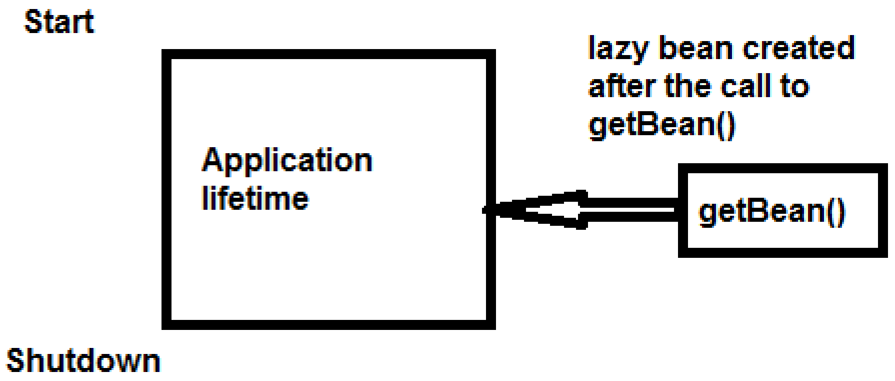
```
<bean id="bean1" class="Bean1"/>
<bean id="bean2" class="Bean2" lazy-init="false"/>
<bean id="bean3" class="Bean3" lazy-init="default"/>
<bean id="bean4" class="Bean4" lazy-init="true"/>
```

Lazy initialization of all beans in a container:

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

Lazy and Not Lazy Beans Lifetime





Task: change the XML configuration to use lazy initialization for all beans, by default.

ex.6

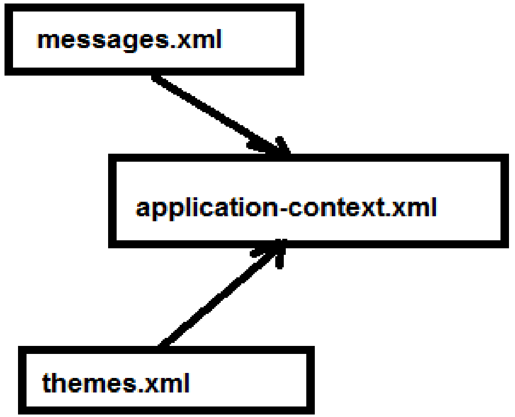
Spring Framework – Context Import

It’s often convenient to break the context into several files:

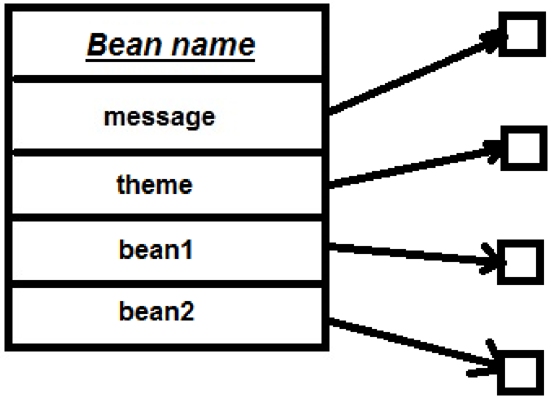
```
<beans>
   <import resource="messages.xml"/>
   <import resource="themes.xml"/>

   <bean id="bean1" class="com.luxoft.springioc.example07.Bean1"/>
   <bean id="bean2" class="com.luxoft.springioc.example07.Bean2"/>
</beans>
```

Spring Framework – Context Import

Context import functionality:



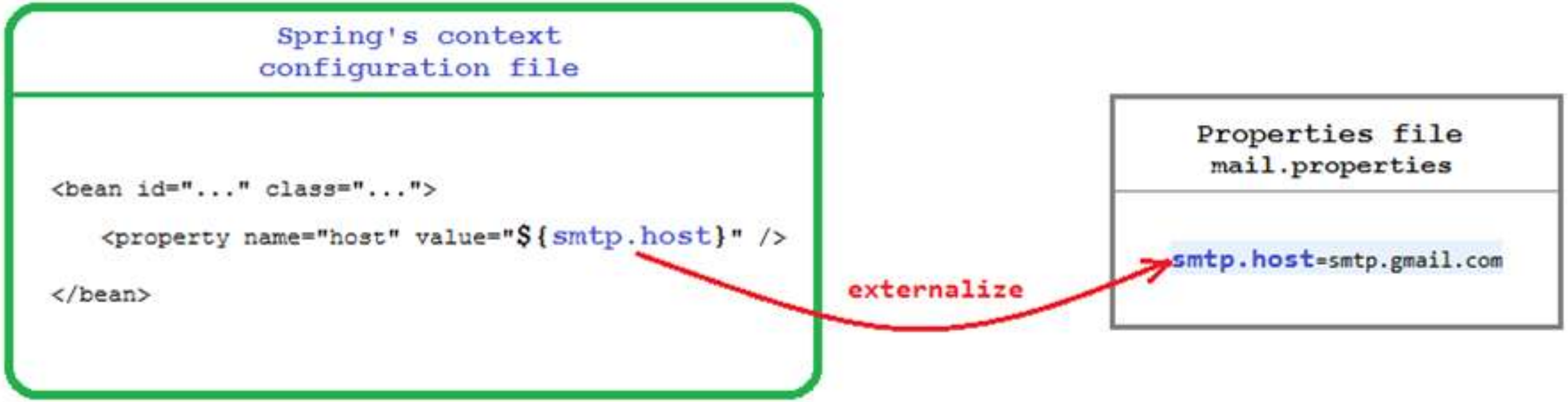
Context content:



ex.7

Spring Framework – Use of Property Files with Context

- Spring allows to externalize literals in its context configuration files into external properties
- In Spring, context configuration files use placeholders: `${variable_name}`
- Spring reads properties files declared by `PropertyPlaceholderConfigurer` bean



```
<bean id="..." class="...">
    <property name="host" value="${smtp.host}" />
</bean>
```

```
smtp.host=smtp.gmail.com
```

Spring Framework – Use of Property Files with Context

- By default, Spring looks for the properties files in the application’s directory

```
<property name="location" value="WEB-INF/jdbc.properties" />
```

It will find the jdbc.properties file under WEB-INF directory of the application (in case of a Spring MVC application)

- We can use the prefix `classpath:` to tell Spring to load a properties file in the application’s `classpath`

```
<property name="location" value="classpath:jdbc.properties" />
```

- Use the prefix `file:///` or `file:` to load a properties file from an absolute path

```
<property name="location" value="file:///D:/Config/jdbc.properties" />
```

Spring Framework – Use of Property Files with Context

```xml
<bean class="PropertyPlaceholderConfigurer">
   <property name="locations" value="classpath:example08/jdbc.properties"/>
</bean>

<bean id="dataSource" class="com.luxoft.springioc.example08.DataSource">
   <property name="driverClassName" value="${jdbc.driverClassName}" />
   <property name="url" value="${jdbc.url}" />
   <property name="username" value="${jdbc.username}" />
   <property name="password" value="${jdbc.password}" />
</bean>
```

jdbc.properties:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=password
```

ex.8

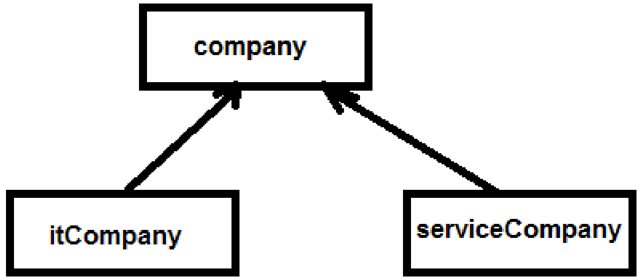
Spring Framework – Use of Alias

The bean named **originalName** may be referred as **aliasName**

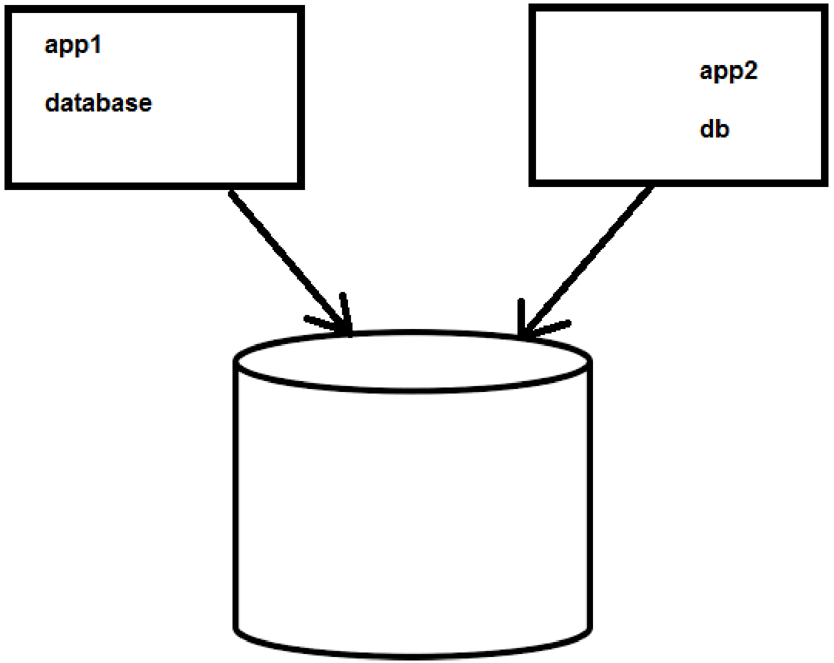
It is used to provide future bean specialization. For example, we may refer beans as **serviceCompany** and **itCompany**, but for a while we have no special implementation for it, we use aliases:

```
<bean id="company" class="com.luxoft.springioc.example09.Company"/>

<alias name="company" alias="itCompany"/>
<alias name="company" alias="serviceCompany"/>
```

Spring Framework – Use Cases for Alias



Bean specialization



Override bean definitions inherited from external sources

ex.9

Spring Framework – Constructor Dependency Injection

Dependency injection with use of constructor with arguments:

```java
public class Company {
    private String name;

    public Company(String name) {
        this.name = name;
    }
    …
}
```

```java
public class Person {
    private String name;
    private Company company;

    public Person(String name, Company company) {
        this.name = name;
        this.company = company;
    }
    …
}
```

Spring Framework – Constructor Dependency Injection

```
<bean id="luxoftCompany" class="com.luxoft.springioc.example10.Company" >
      <constructor-arg value="Luxoft" />
</bean>

<bean id="smithPerson" class="com.luxoft.springioc.example10.Person">
     <constructor-arg value="John Smith" />
      <constructor-arg ref="luxoftCompany" />
</bean>
```

ex.10

Spring Framework – Constructor Dependency Injection

Cyclic dependency:

```
public class A {
   private B b;

   public A(B b) {
     this.b = b;
   }

}
```

```
public class B {
   private A a;

   public B(A a) {
     this.a = a;
   }

}
```

We will get ***BeanCurrentlyInCreationException*** during Dependency Injection
Solution: to replace Constructor Dependency Injection with Setter Dependency Injection in one or both classes

ex.11_error

Replace Constructor Injection with Setter Injection

```java
public class A {
   private B b;

   public B getB() {
     return b;
   }

   public void setB(B b) {
     this.b = b;
  }
}
```

```java
public class B {
    private A a;

    public A getA() {
        return a;
    }

    public void setA(A a) {
        this.a = a;
    }
}
```

```xml
<bean id="a" class="com.luxoft.springioc.example11_correct.A">
    <property name = "b" ref="b"/>
</bean>
<bean id="b" class="com.luxoft.springioc.example11_correct.B">
    <property name = "a" ref="a"/>
</bean>
```

ex. 11_correct

Spring Framework – Setter Dependency Injection

```java
public class Person {
    private Company company;
    private String name;
    ...

    public void setCompany(Company company) {
       this.company = company;
    }
}
```

```xml
<bean id="luxoftCompany" class="com.luxoft.springioc.example12.Company" >
   <property name="name" value="Luxoft" />
</bean>

<bean id="smithPerson" class="com.luxoft.springioc.example12.Person">
    <property name="name" value="John Smith" />
    <property name="company" ref="luxoftCompany" />
</bean>
```

ex.12

Spring Framework – Autowiring

Spring is able to resolve and add dependencies automatically

```
<bean id="..." class="..." autowire="no|byName|byType|constructor" />
```

- Can cause configuration to keep itself up to date
- It can significantly reduce the volume of configuration

- Autowiring by type can only work if application context contains exactly one bean of a property type
- It is harder to read and check dependencies

Spring Framework – Autowiring

Autowiring modes:

– **no**: no autowiring. This is the default

– **byName**: container looks 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**: container looks for a bean of specific class, works only if there is exactly one bean of property type in container otherwise **UnsatisfiedDependencyException** is thrown

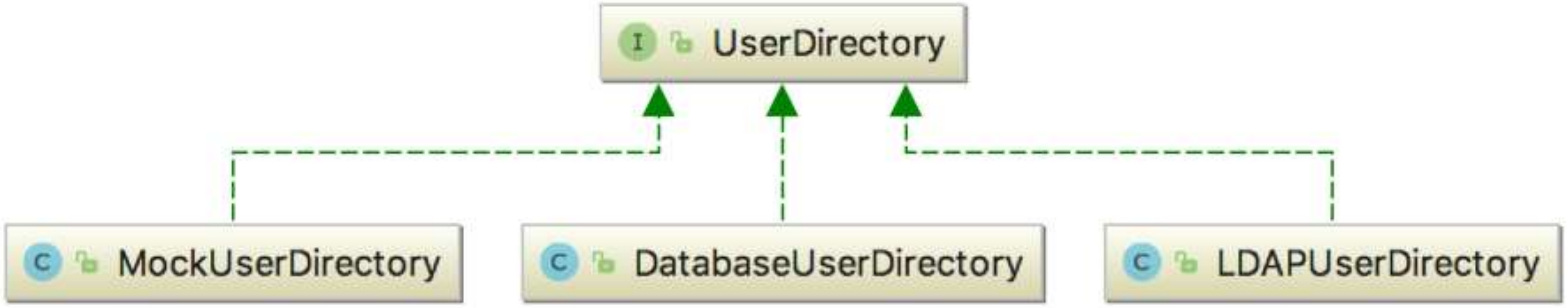
– **constructor**: will create object using constructor and use **byType** autowiring to find arguments

Spring Framework – Autowiring

If there is more than one bean of a given type and we try to autowire byType, we are getting an error like the following:

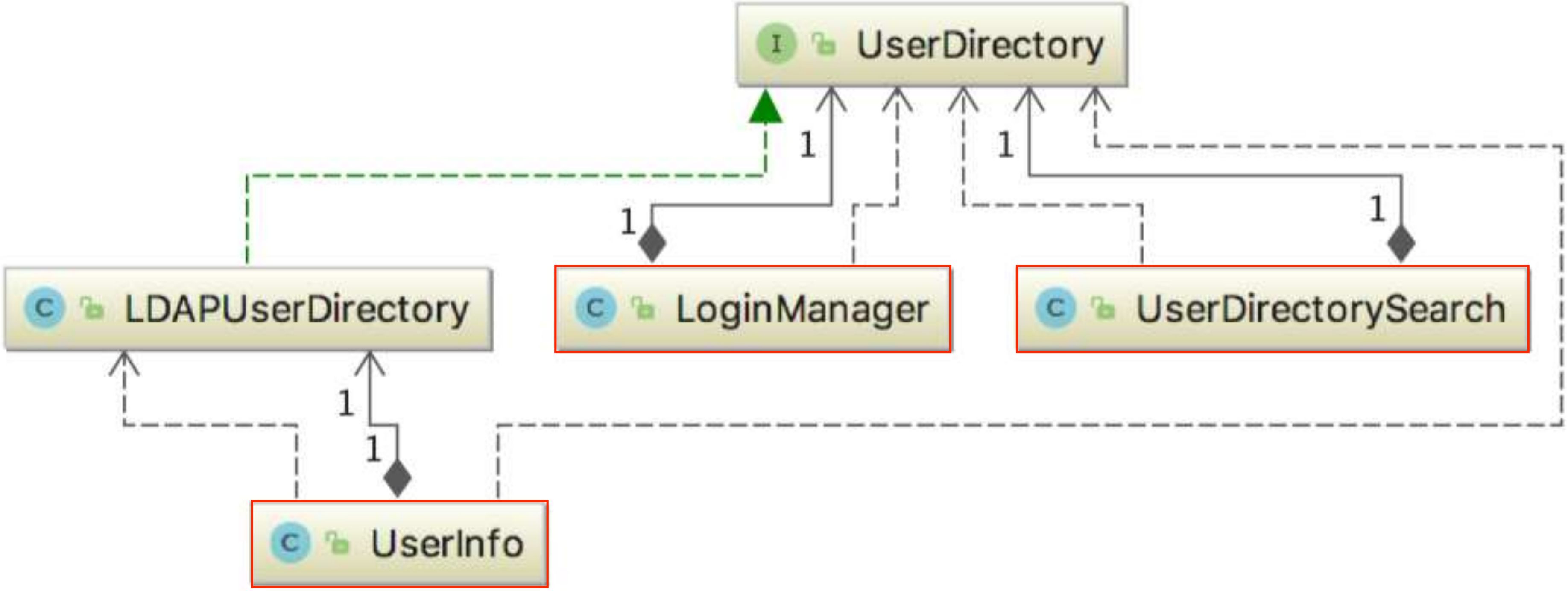
```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean
with name 'userInfo' defined in class path resource [example14/application-context.xml]:
Unsatisfied dependency expressed through bean property 'userDirectory':

No qualifying bean of type [com.luxoft.springioc.example14.UserDirectory] is defined:
expected single matching bean but found 2: userDirectory,userDirectory2;

nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type [com.luxoft.springioc.example14.UserDirectory] is defined:
expected single matching bean but found 2: userDirectory,userDirectory2
```

Spring Framework – Autowiring



ex.13

Spring Framework – Autowiring



ex.13

Spring Framework – Autowiring

Let’s have classes which need the information about the user:

```
<bean id="userDirectory" class="com.luxoft.springioc.example13.LDAPUserDirectory" />

<bean id="loginManager" class="com.luxoft.springioc.example13.LoginManager">
   <property name="userDirectory" ref="userDirectory" />
</bean>

<bean id="userDirectorySearch" class="com.luxoft.springioc.example13.UserDirectorySearch">
   <property name="userDirectory" ref="userDirectory" />
</bean>

<bean id="userInfo" class="com.luxoft.springioc.example13.UserInfo">
   <property name=“ldapUserDirectory" ref="userDirectory" />
</bean>
```

ex.13

Spring Framework – Autowiring

Now let’s turn on the autowiring:

```
public class LoginManager {
   private UserDirectory userDirectory;
}

public class UserDirectorySearch {
   private UserDirectory userDirectory;
}

public class UserInfo {
   private LDAPUserDirectory
           ldapUserDirectory;
}
```

```
<bean id="userDirectory"
      class="LDAPUserDirectory" />


<bean id="loginManager" class="LoginManager"
      autowire="byName" />


<bean id="userDirectorySearch"
      class="UserDirectorySearch"
      autowire="byName" />


<bean id="userInfo"
      class="UserInfo"
      autowire="byType" />
```

ex.14

Tasks for ex. 13 and 14

- Change example 13 so that it is using autowiring
- Change example 14 so that the **`userInfo`** bean has autowiring **`byName`** instead of **`byType`**; explain the difference in execution
- Change example 14 so that the **`UserInfo`** class contains a **`UserDirectory`** type field instead of **`LDAPUserDirectory`**; execute the program and note that autowiring is permitted also for classes that are descendants of a given class

Spring Framework – Collections Initialization

```java
public class Customer {
    private List<Object> list;
    …
}
```

```xml
<bean id="customerBean" class="com.luxoft.springioc.example15.Customer">
    <!-- java.util.List -->
    <property name="list">
        <list>
            <value>1</value>
            <ref bean="personBean" />
            <bean class="com.luxoft.springioc.example15.Person">
                <property name="name" value="John" />
                <property name="address" value="address" />
                <property name="age" value="28" />
            </bean>
        </list>
    </property>
```

Spring Framework – Collections Initialization

```
public class Customer {
    …
    private Set<Object> set;
}

    <!-- java.util.Set -->
    <property name="set">
        <set>
            <value>1</value>
            <ref bean="personBean" />
            <bean class="com.luxoft.springioc.example15.Person">
                <property name="name" value="John" />
                <property name="address" value="address" />
                <property name="age" value="28" />
            </bean>
        </set>
    </property>
```

Spring Framework – Collections Initialization

```
public class Customer {
    …
    private Map<Object, Object> map;
}

    <!-- java.util.Map -->
    <property name="map">
        <map>
            <entry key="Key 1" value="1" />
            <entry key="Key 2" value-ref="personBean"/>
            <entry key="Key 3">
                <bean class="com.luxoft.springioc.example15.Person">
                   <property name="name" value="John" />
                   <property name="address" value="address" />
                   <property name="age" value="28" />
                </bean>
            </entry>
        </map>
    </property>
```

Spring Framework – Collections Initialization

The same as:

```
Customer customerBean = (Customer)context.getBean("customerBean");
customerBean.getMap().put("Key 1", "1");
customerBean.getMap().put("Key 2", context.getBean("personBean");


Person person = new Person();
person.setName("John");
person.setAddress("address");
person.setAge(28);
customerBean.getMap().put("Key 3", person);
```

Spring Framework – Collections Initialization

```java
public class Customer {
    …
    private Map<String, Object> stringsMap;
    …
}
```

```xml
<!-- java.util.Map -->
<property name="stringsMap">
    <map>
        <entry key="String key 1" value="1" />
        <entry key="String key 2" value-ref="personBean"/>
    </map>
</property>
```

Spring Framework – Collections Initialization

```
public class Customer {
    …
    private Map<Person, String> personsMap;
    …
}

<!-- java.util.Map -->
<property name="personsMap">
    <map>
        <entry key-ref="personBean" value="USA" />
    </map>
</property>
```

Spring Framework – Collections Initialization

```java
public class Customer {
    …
    private Properties props;
}
```

```xml
    <!-- java.util.Properties -->
    <property name="props">
        <props>
            <prop key="admin">admin@example.com</prop>
            <prop key="support">support@example.com</prop>
        </props>
    </property>
```

ex.15

Spring Framework – Properties Inheritance

```
<bean id="testBean"
      abstract="true" class="com.luxoft.springioc.example16.TestBean">
    <property name="name" value="parent" />
    <property name="age" value="1" />
</bean>


<bean id="inheritsWithDifferentClass"
      class="com.luxoft.springioc.example16.DerivedTestBean" parent="testBean">
    <property name="name" value="override" />
    <!-- the age property value of 1 will be inherited from parent -->
</bean>
```

ex.16

Task for ex. 16

- Change example 16 so that the parent class is no longer **`abstract`**; make sure that you make the modifications both at the level of the class and configuration

Spring Framework – Merge of Collections

```
<bean id="parent" abstract="true" class="ComplexObject">
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.com</prop>
            <prop key="support">support@example.com</prop>
        </props>
    </property>
</bean>
<bean id="child" parent="parent">
    <property name="adminEmails">
    <!-- the merge is specified on the *child* collection definition -->
        <props merge="true">
            <prop key="sales">sales@example.com</prop>
            <prop key="support">support@example.co.uk</prop>
        </props>
    </property>
</bean>
```

```
administrator=administrator@example.com
sales=sales@example.com
support=support@example.co.uk
```

ex.17

Applicable to properties, list, set, map.

Spring Framework – Empty and Null Properties

```xml
<bean id="exampleBean"
      class="com.luxoft.springioc.example18.ExampleBean">
   <property name="name" value="" />
   <property name="email"><null/></property>
</bean>
```

ex.18

Spring Framework – p-namespace

```
<bean id="classic" class="com.luxoft.springioc.example19.ExampleBean">
   <property name="email" value="foo@bar.com" />
</bean>

<bean id="p-namespace" class="com.luxoft.springioc.example19.ExampleBean"
                         p:email="foo@bar.com" />

<bean id="john-classic" class="com.luxoft.springioc.example19.Person">
   <property name="name" value="John Doe" />
   <property name="spouse" ref="jane" />
</bean>

<bean id="john-modern" class="com.luxoft.springioc.example19.Person"
      p:name="John Doe“ p:spouse-ref="jane" />

<bean id="jane" class="com.luxoft.springioc.example19.Person">
   <property name="name" value="Jane Doe" />
</bean>
```

ex.19

Exercise

Lab guide:

- Exercise 1