Spring Framework
Inversion of control

Part 1

LUXOFT TRAINING

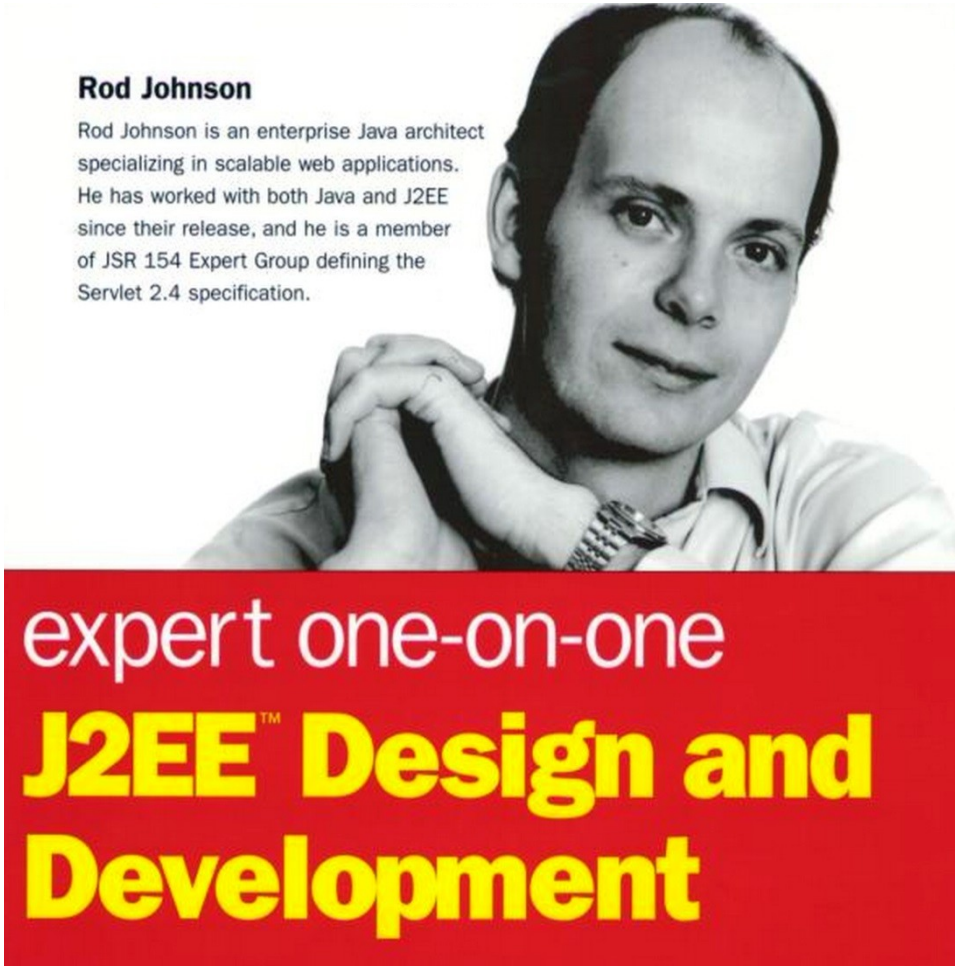
LXFT LISTED NYSE

Spring Framework - Introduction

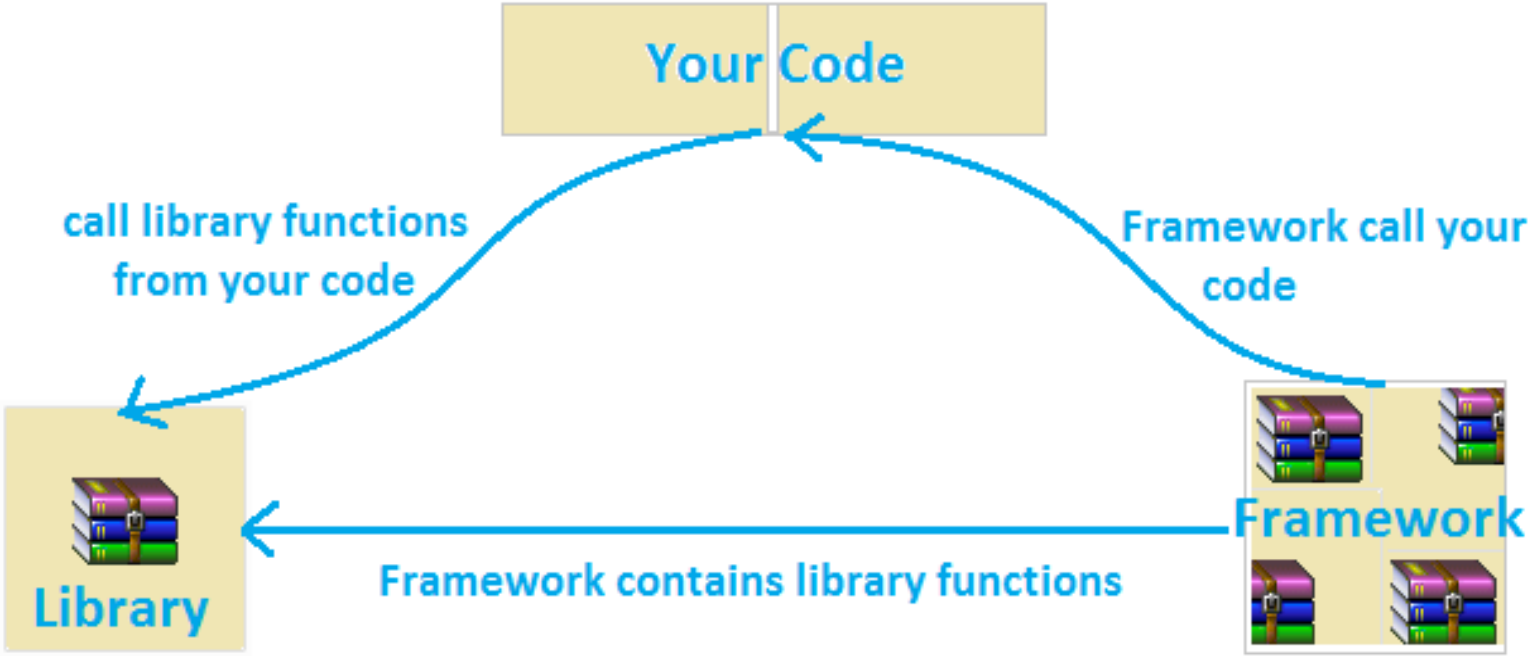
- Spring is a **lightweight**, but at the same time **flexible** and **universal** framework used for creating Java SE and Java EE applications
- Spring is a framework with an **open source code**
- Spring is an **application framework**, not a layer framework
- Spring includes several separate frameworks

Spring Framework - Introduction

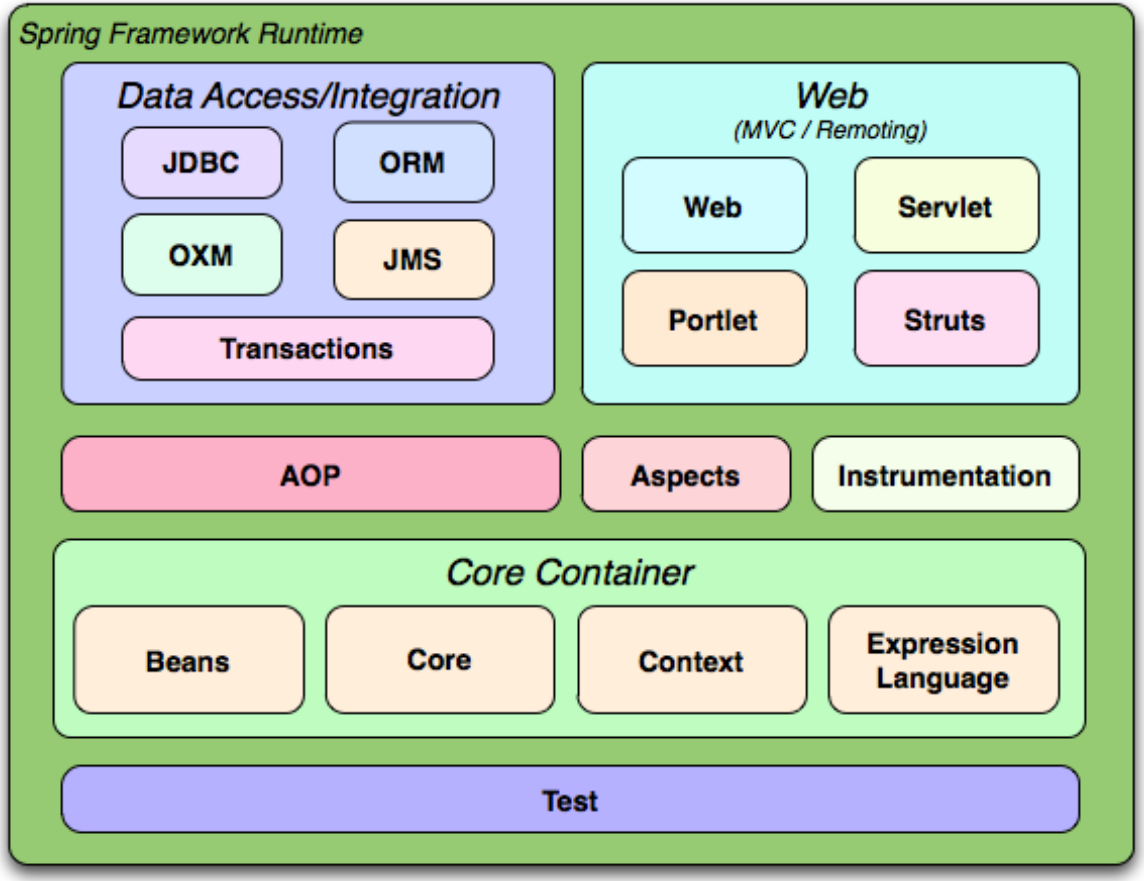
- **Rod Johnson** created Spring in **2003**
- Spring took its rise from books ***Expert One-on-One Java J2EE Design and Development*** and ***J2EE Development Without EJB***
- The basic idea behind Spring is to **simplify** traditional approach to designing J2EE applications


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

Difference between library and framework



Spring Framework - Framework structure



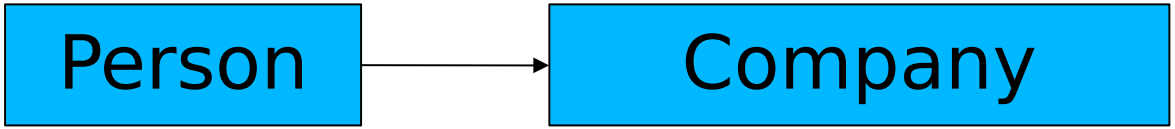
Spring Framework - Framework structure

- **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**.
- This capability applies to the Java SE programming model and to full and partial Java EE.

Spring Framework – Object dependencies

- Spring implements various design patterns
  - ***Factory***
  - ***Abstract Factory***
  - ***Builder***
  - ***Proxy***
- The Spring Framework **Inversion of Control (IoC)** provides a formalized means of composing disparate components into a fully working application.

Spring Framework – Object dependencies



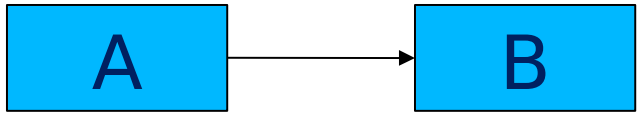
Traditional approach

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

   public Person() {
      name = "John Smith";
      company = new Company();
      company.setName("Luxoft"
      );
   }
   }
```

```
public class Company {
   private String name;

   public void setName(String
   name) {
       this.name = name;
   }
   }
```

Spring Framework – Object dependencies

Traditional approach



Problems:

- Class A directly depends on class B
- It is impossible to test A separately from B
- The lifetime of object B depends on A – it is impossible to use B object in other places
- It is not possible to replace B by another implementation

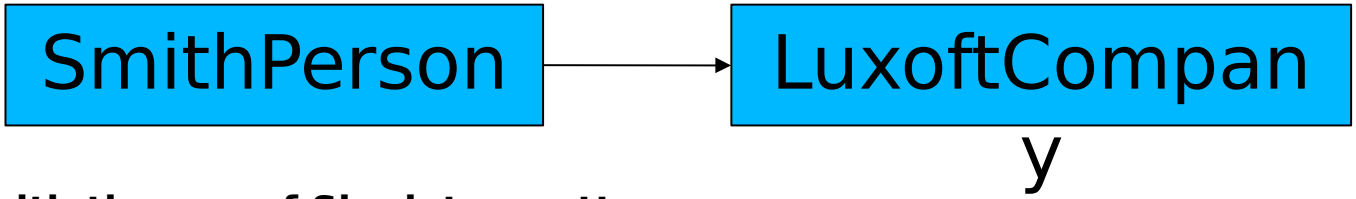
Spring Framework – Object dependencies

Approach with the use of Singleton pattern

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

   public String getName()
   {
      return name;
   }
   …
   }
```

```
public class Company {
   private String name;

   public String getName()
   {
       return name;
   }
   …
   }
```

Spring Framework – Object dependencies

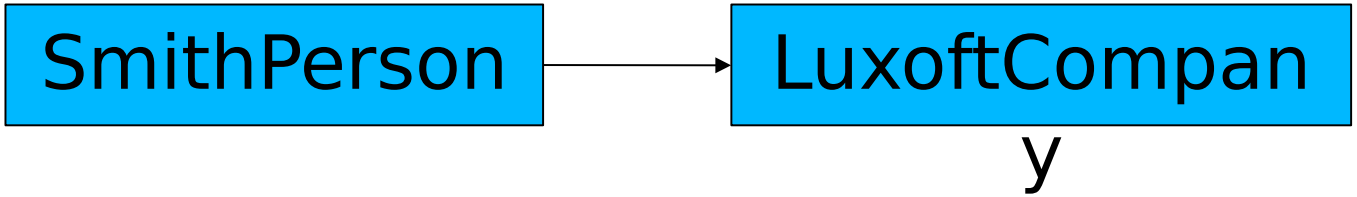


Approach with the use of Singleton pattern

```
public class SmithPerson extends Person
{
   private static Person smithPerson =
      new Person();

   public static Person getPerson() {
      smithPerson.setName("John Smith");
      smithPerson.setCompany(
           LuxoftCompany.getCompany());
      return smithPerson;
   }
   }
```

```
public class LuxoftCompany
      extends Company {
   private static Company
luxoftCompany
         = new Company();

   public LuxoftCompany() {
      luxoftCompany.setName("Luxoft");
   }

   public static Company getCompany()
   {
      return luxoftCompany;
   }
   }
```

Spring Framework – Object dependencies

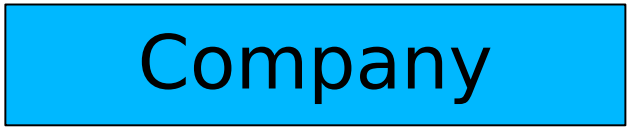


- Separate class specially for our task
- There’s a direct link to the **LuxoftCompany** in **SmithPerson**
- In case of the transfer of Smith to another company, we have to change a piece of code
- It’s impossible to temporarily “replace” company for testing

Spring Framework – Object dependencies

- **Dependency Injection (DI)** is also known as **Inversion of Control (IoC)**.
- Objects define their dependencies, that is, the other objects they work with.
- **The container then injects those dependencies** when it creates the bean. **This process is fundamentally the inverse**, hence the name **Inversion of Control (IoC)**.

Spring Framework – Object dependencies



Inversion of Control Container approach

POJO – Plain Old Java Object

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

    public class Company {
    private String name;
    }

    public class
    CompanyReport {
        private Company
    company;
```

application-context.xml

```
<bean id="smith" class="Person">
    <property name="name" value="John Smith"/>
    <property name="company" ref="luxoftCompany"/>
    </bean>

    <bean id="luxoftCompany" class="Company">
        <property name="name" value="Luxoft"/>
    </bean>

    <bean id="companyReport"
    class="CompanyReport">
        <property name="company"
    ref="luxoftCompany"/>
```

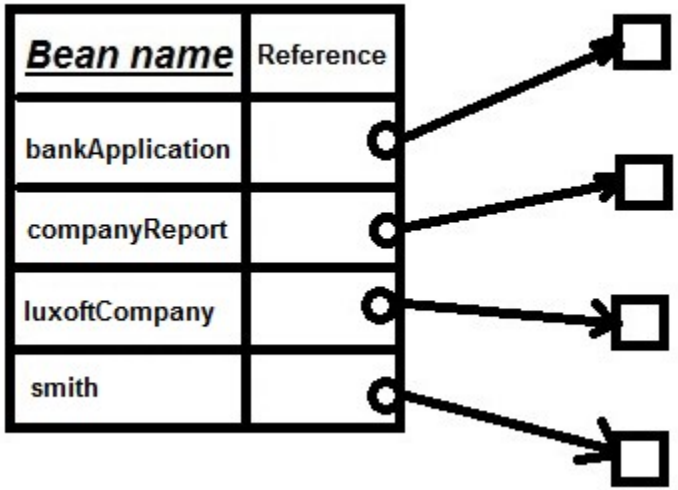
Spring Framework – Object dependencies

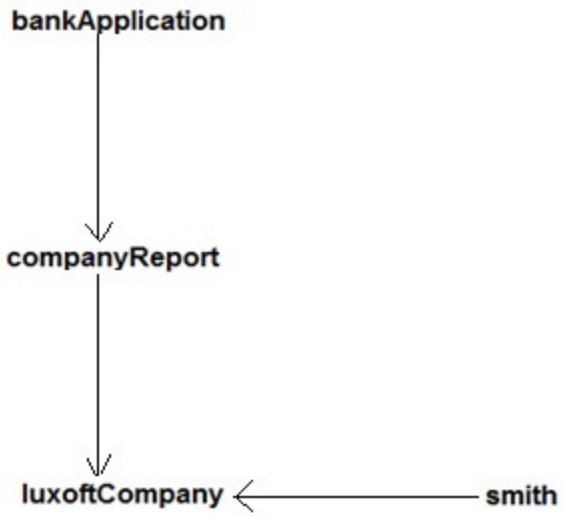
```
<bean id="bankApplication" class="BankApplication">
    <property name="companyReport" ref="companyReport "/>
</bean>

public class BankApplication {
    private CompanyReport companyReport;
    }

    public class Main {
    …
    public static void main(String args[]) {
        ClassPathXmlApplicationContext context = new
        ClassPathXmlApplicationContext("dependencies/application-context.xml");
        BankApplication bankApplication = (BankApplication)
                        context.getBean(“bankApplication");
        System.out.println(bankApplication.getCompanyReport().getCompany().getName
        ());
        context.close();
    }
    }
```

ex. dependencies

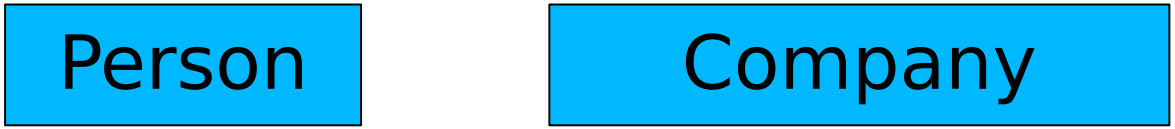
Internal structure of the application context

- 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



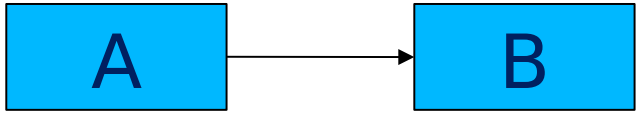
Inversion of Control Container approach

Advantages:

- The container creates the necessary objects and manages their lifetime
- ***Person*** and ***Company*** are not dependent and do not depend on any outer libraries
- ***application-context*** documents the system and objects dependencies
- It’s very easy to make changes to object dependencies in the system

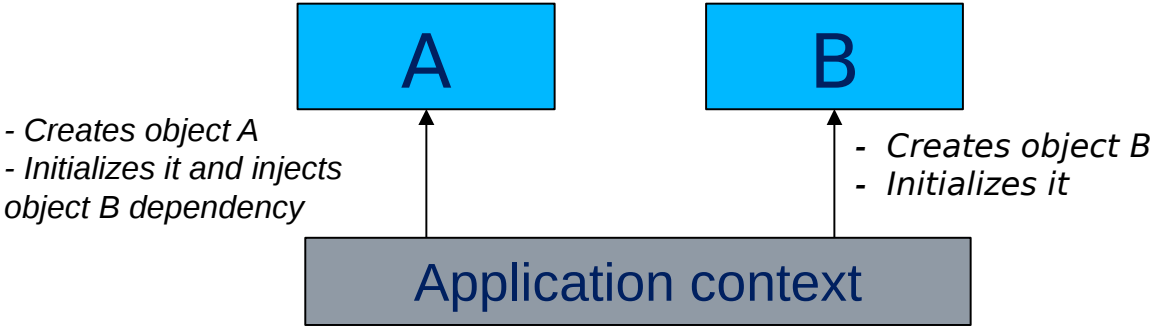
Spring Framework – Object dependencies

Traditional approach: dependencies inside the code




```
class A {
    private B
    b;
    }

class B {

}
```

IoC: objects know nothing about each other



Spring Framework – IoC / DI

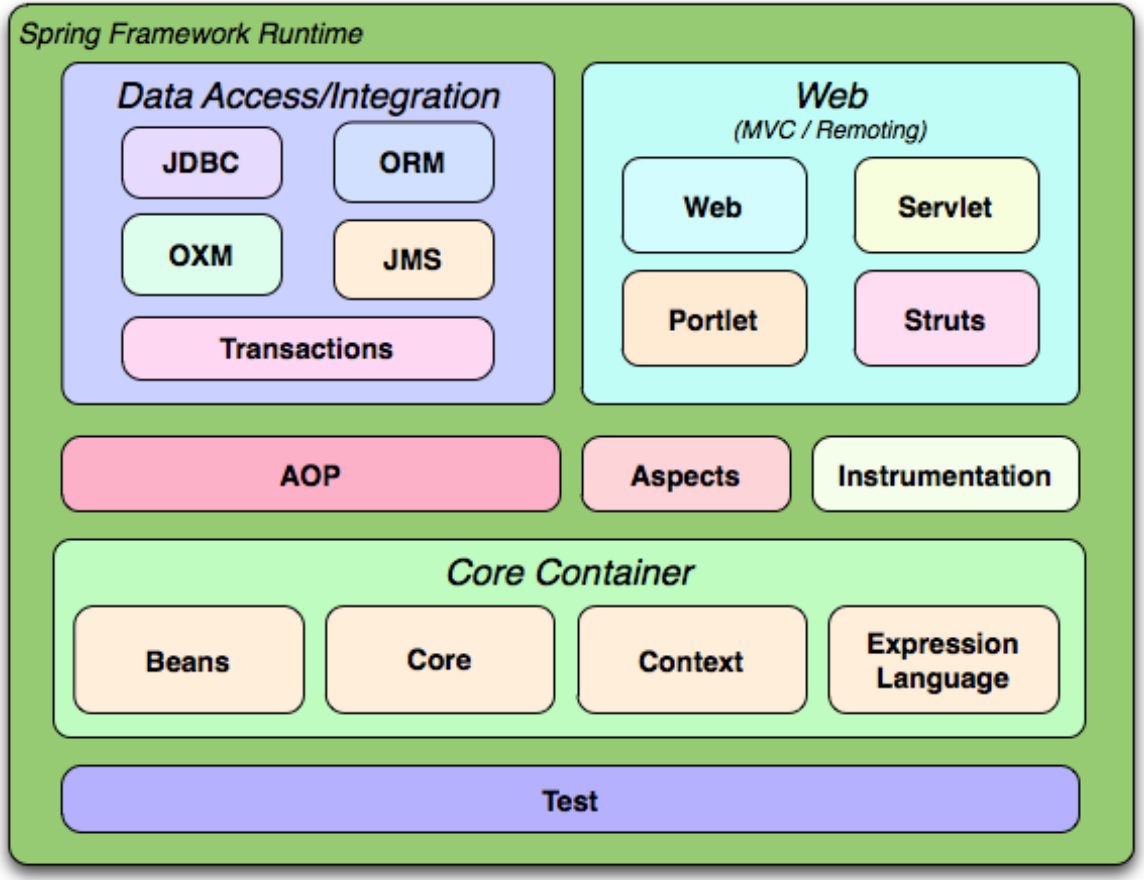
- Inversion of Control (IoC) pattern is the base for Spring
  - “Hollywood Principle” – Don't call me, I'll call you
  - The basic idea is to eliminate the dependency of application components from certain implementation and to delegate IoC container rights to control classes instantiation
  - Martin Fowler suggested the name of Dependency Injection (DI) because it better reflects the essence of the pattern (http://www.martinfowler.com/articles/injection.html)

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 IoC container

Spring Framework - Core Container



Core Container consists of:

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

Spring Framework – IoC containers

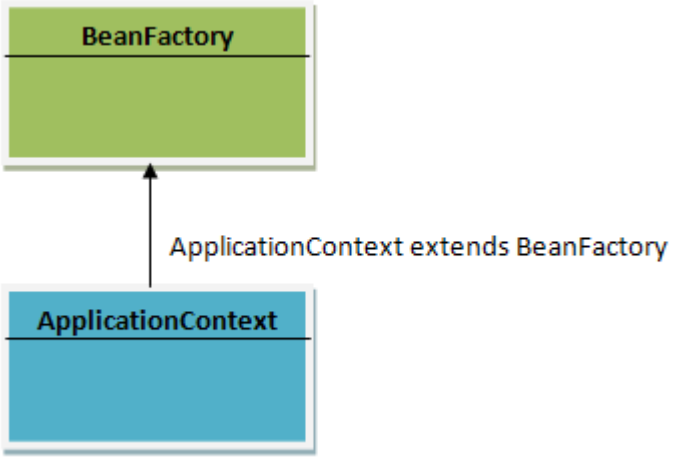
- The ***BeanFactory*** is a central **IoC** container interface into the **Spring Framework**
- Implementation of the factory pattern
- Most common implementation: ***XmlBeanFactory***
- ***BeanFactory*** provides only basic low-level functionality

Spring Framework – IoC containers

- **`ApplicationContext extends BeanFactory`** and adds:
  - Event handling
  - Internationalization
  - Work with resources and messages
  - Simple integration with Spring AC
  - 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

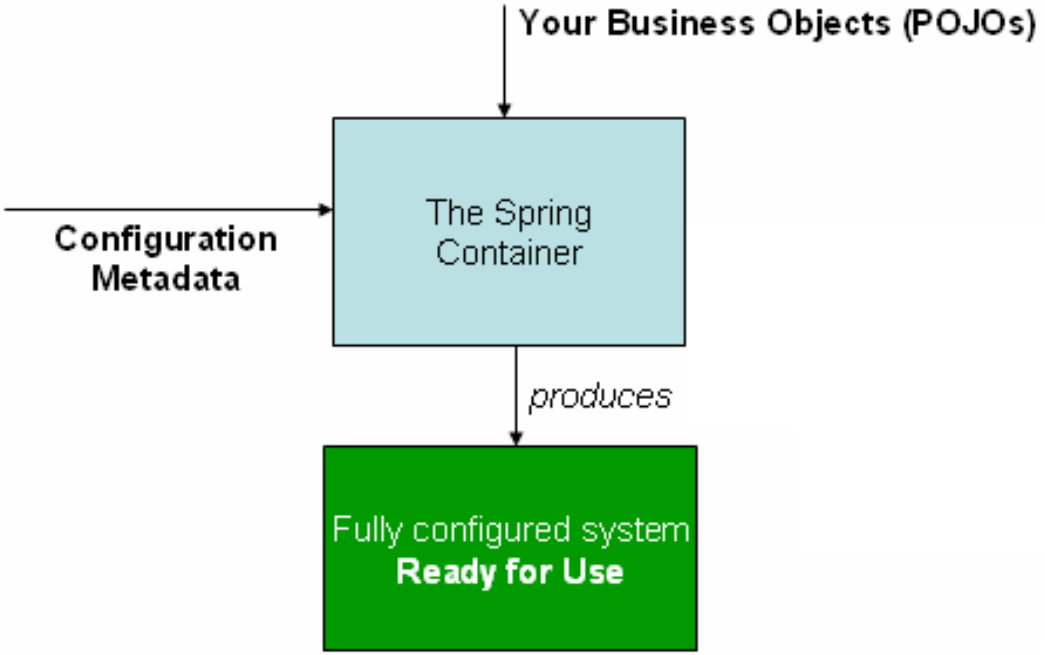
- Most widely used implementations of **ApplicationContext**:
  - **GenericXmlApplicationContext** (since v.3.0)
  - **ClassPathXmlApplicationContext**
  - **FileSystemXmlApplicationContext**
- XML is a traditional way to configure a container
- It is easier and faster to use annotation-based configuration, 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 Spring IoC container can be represented as follows:



Spring Framework – Maven configuration

```
<!-- Spring framework -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.7.RELEASE</version>
</dependency>
```

Spring Framework – Working with IoC container

Container creation:

```
ClassPathXmlApplicationContext context =
    new ClassPathXmlApplicationContext("application-
context.xml");
Bean1 bean1 = (Bean1)context.getBean("bean1");
```

ex.1

Spring Framework – Working with IoC container

```
ClassPathXmlApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});
ServiceBean serviceBean = (ServiceBean)context.getBean("serviceBean");
DaoBean daoBean = (DaoBean)context.getBean("daoBean");
Bean1 bean1 = (Bean1)context.getBean("bean1");
```

services.xml

```
<bean id="serviceBean" class="ServiceBean"/>
<bean id="bean1" class="Bean1">
   <property name="name" value="bean12“/>
</bean>
```

daos.xml

```
<bean id="daoBean" class="DaoBean"/>
<bean id="bean1" class="Bean1">
   <property name="name" value="bean11"/>
</bean>
```

ex. 2

Task for ex. 1 and 2

- Fix the execution of the example01/Tutor.java test
- Note the definition of 2 “bean1” objects into the XML configuration of example02. Change the example so that, instead of *bean11*, it prints *bean12*

Spring Framework – Bean creation

With the use of no-args constructor:

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

ex.3

Spring Framework – Bean creation

With the use of a 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 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

With the use of not static factory method:

```
<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 to the time it is first addressed.

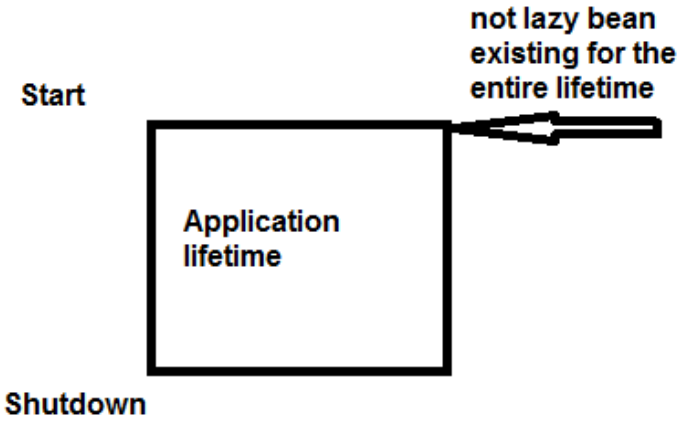
Lazy initialization of single bean:

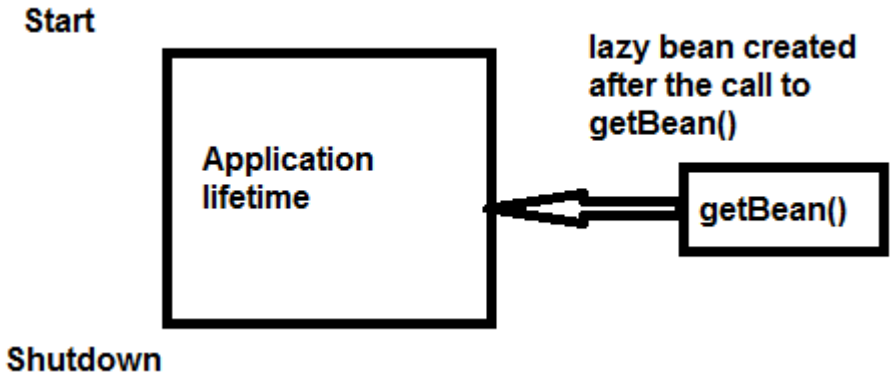
```
<bean id="bean1" class="Bean1"/>
<bean id="bean2" class="Bean2" lazy-init="false"/>
<bean id="bean3" class="Bean3" lazy-init="default"/>
<bean id="bean4" class="Bean4" lazy-init="true"/>
```

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

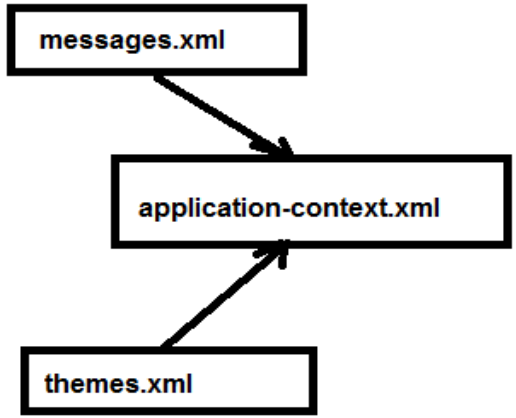
Spring Framework – Context import

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

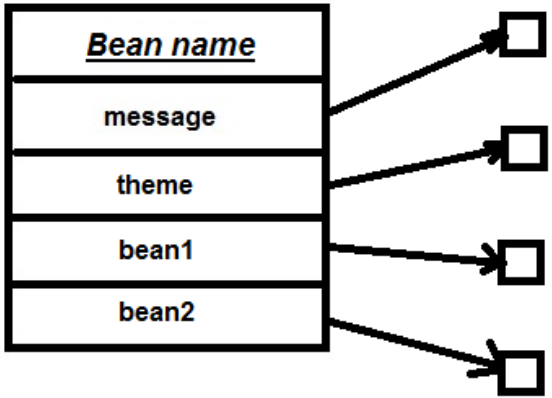
```
<beans>
   <import resource="messages.xml"/>
   <import resource="themes.xml"/>

   <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



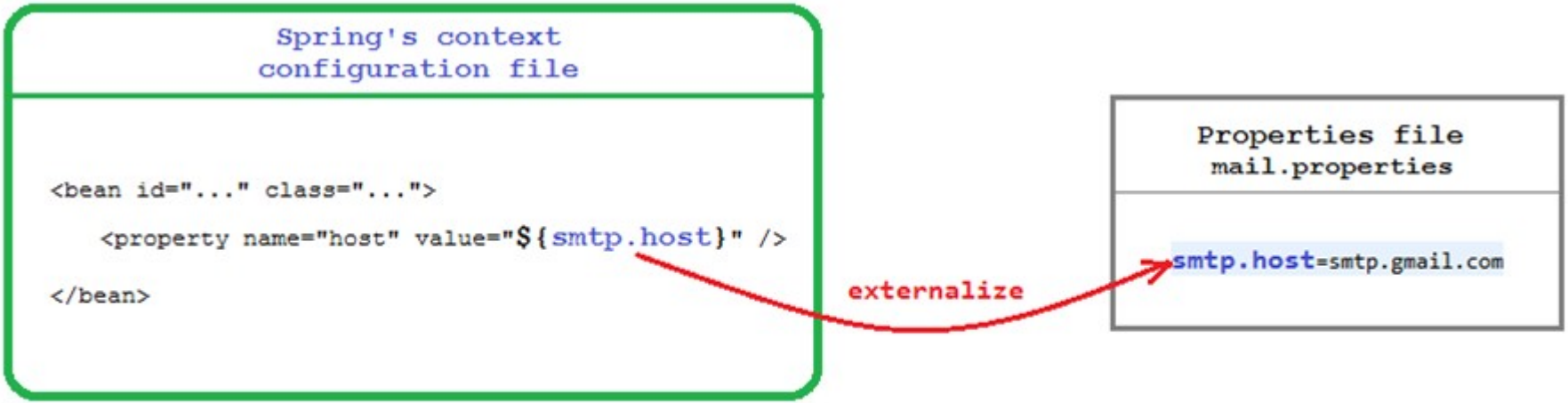
Context content:



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 file use placeholders: `${variable_name}`
- Spring reads properties files declared by



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

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

ex.8

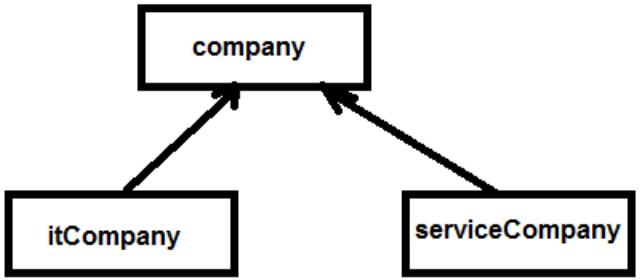
Spring Framework – Use of alias

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

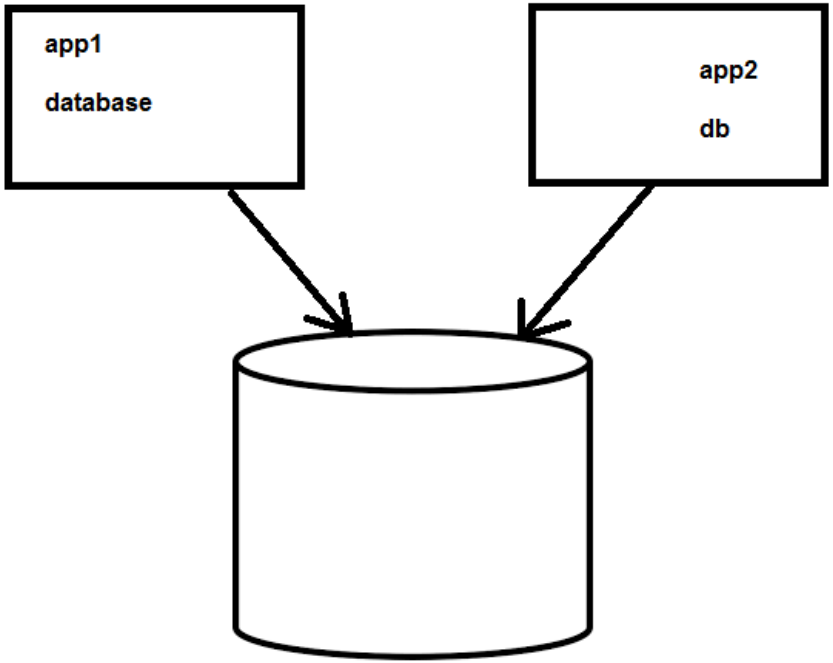
It is used to provide future bean specialization. For example, we may refer beans as **serviceCompany** and **itCompany**, but for a while we have no special implementation for it, we use aliases:

```
<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:

```
public class Company {
    private String name;

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

```
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

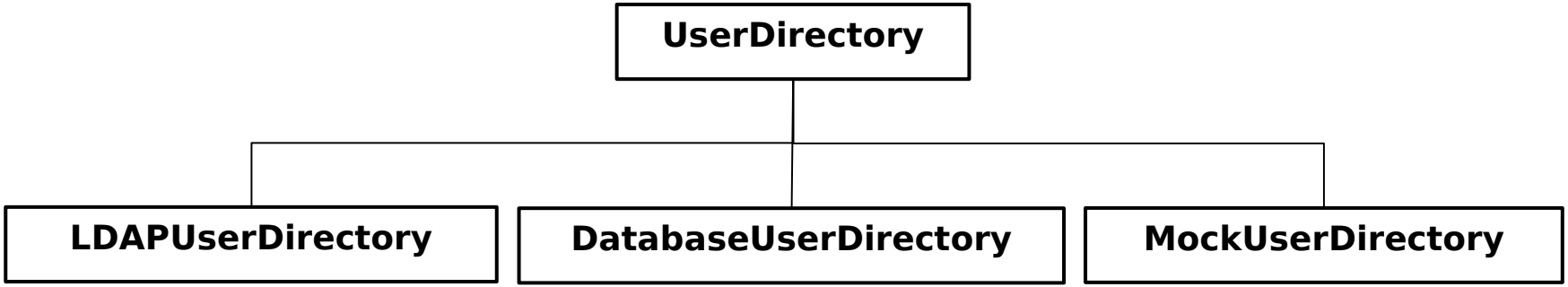
Spring Framework – Setter dependency injection

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

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

   <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

Example: service class to get user info




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

public class UserDirectorySearch {
    private UserDirectory userDirectory;
}

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

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

Spring Framework – Autowiring

- Spring is able to autowire (add dependencies) beans instead of <ref>
- It can significantly reduce the volume of configuration
- Can cause configuration to keep itself up to date
- Autowiring by type can only work if there is exactly one bean of a property type
- It is harder to read and check dependencies

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

Spring Framework – Autowiring

- 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 – 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
```

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

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

   }

   <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

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

    <!-- 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

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


    <!-- 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 of the 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

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

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

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

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

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

ex.19

Exercise

Lab guide:

- Exercise 1