Spring

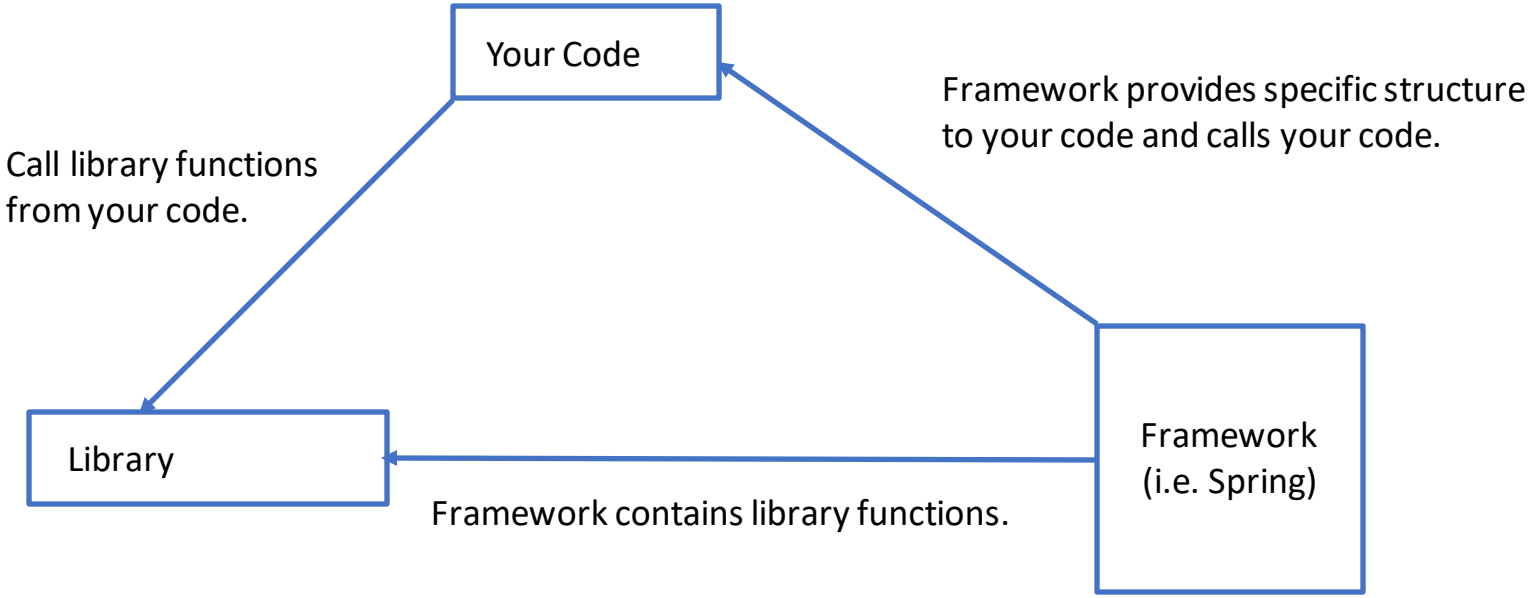
Spring framework

- Spring is an application framework
- Open source
- https://docs.spring.io/spring-framework/docs/current/reference/html/
- Created by Rod Johnson in 2003
- Maintained by Pivotal
- Spring ecosystem includes several separate frameworks: data access, batch processing, cloud applications etc

Spring core

- Main idea: application is a composition of services implemented by POJOs (plain old java objects). Application developer is not required to implement special interfaces like in EJB.
- Spring core is Inversion of Control (IoC) container and library functions relevant to enterprise application development (data access, AOP, web clients, transactions etc)

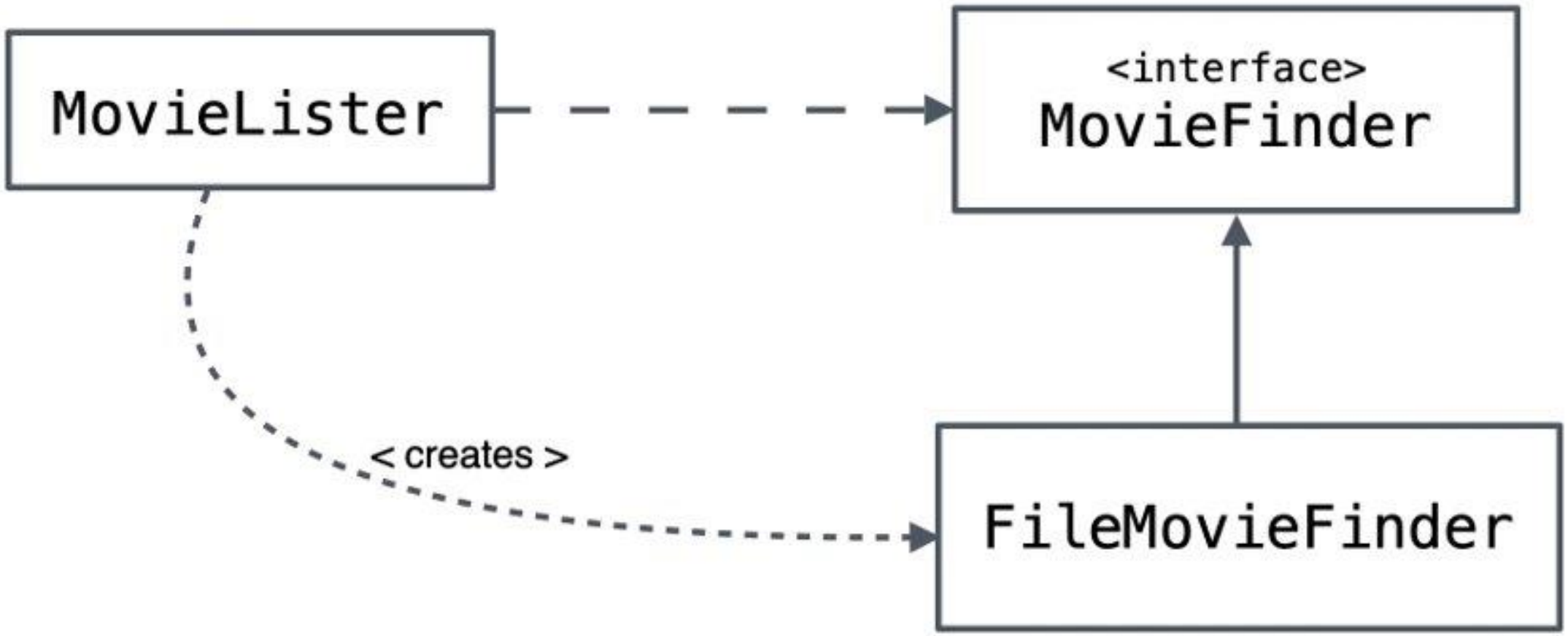
Framework vs library



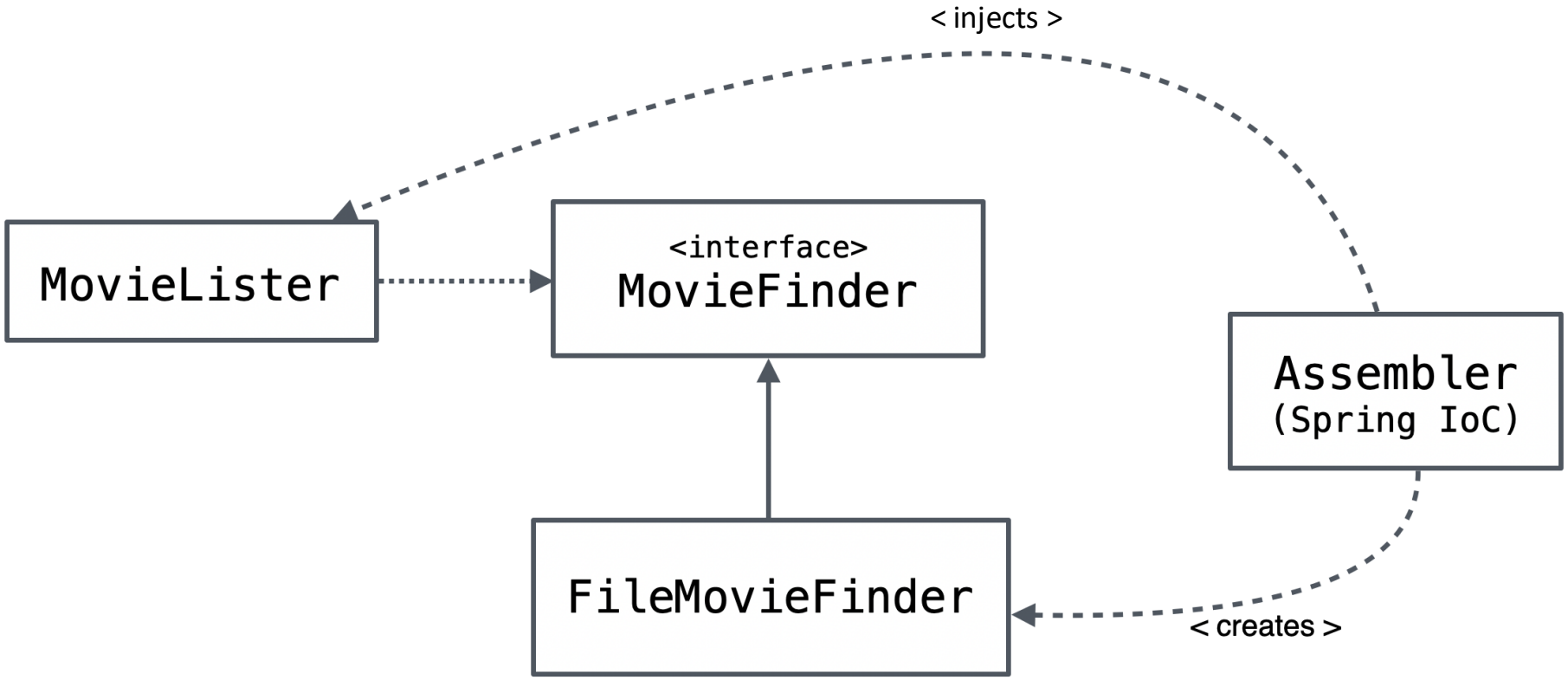
Inversion of control

```
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;
    }
}
```

Owning object creates dependencies



Inversion of control



Inversion of control

- Use different implementations of FileMovieFinder
- Change implementations at runtime
- Use the same instance of FileMovieFinder for several instances of MovieLister or other classes (like Singleton pattern)

Spring Inversion of control

- Object dependencies and configuration by a xml file, annotations or Java config approach.
- Container (Spring IoC) creates and configures objects, then wires dependent objects according to the config

Spring Inversion of control

- Configuration with xml file
- Configuration by annotations was added later

Spring Inversion of control

application-context.xml

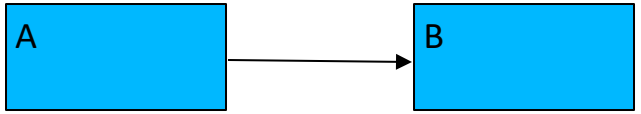
```
<bean id="fileMovieFinder" class="com.luxoft.springioc.movies.FileMovieFinder">
     <property name="fileName" value="movies.txt" />
</bean>

<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");
}
```

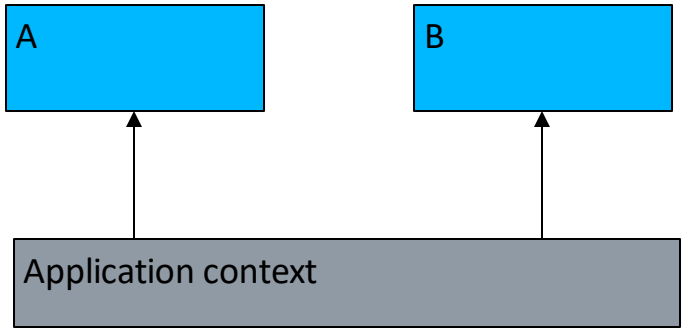
*separate **configuration** from the **code***

Spring Inversion of control

Traditional approach: dependencies inside the code



IoC: objects know nothing about each other



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

Spring Inversion of control

- Dependency management by changing configuration, not the code (only xml)
- Simplifies reusing code
- Simplifies unit testing
- Makes code cleaner

Spring Inversion of control

- Bean – POJO managed (created) by Spring IoC container
- Spring IoC is implemented by ApplicationContext interface
- There are several ApplicationContext implementations
  - **ClassPathXmlApplicationContext**
- XML file contains a series of 'bean' elements that describe Bean definitions
- Bean definitions are used for object creation and configuration as well as dependency wiring

Spring Inversion of control

```
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"/>
```

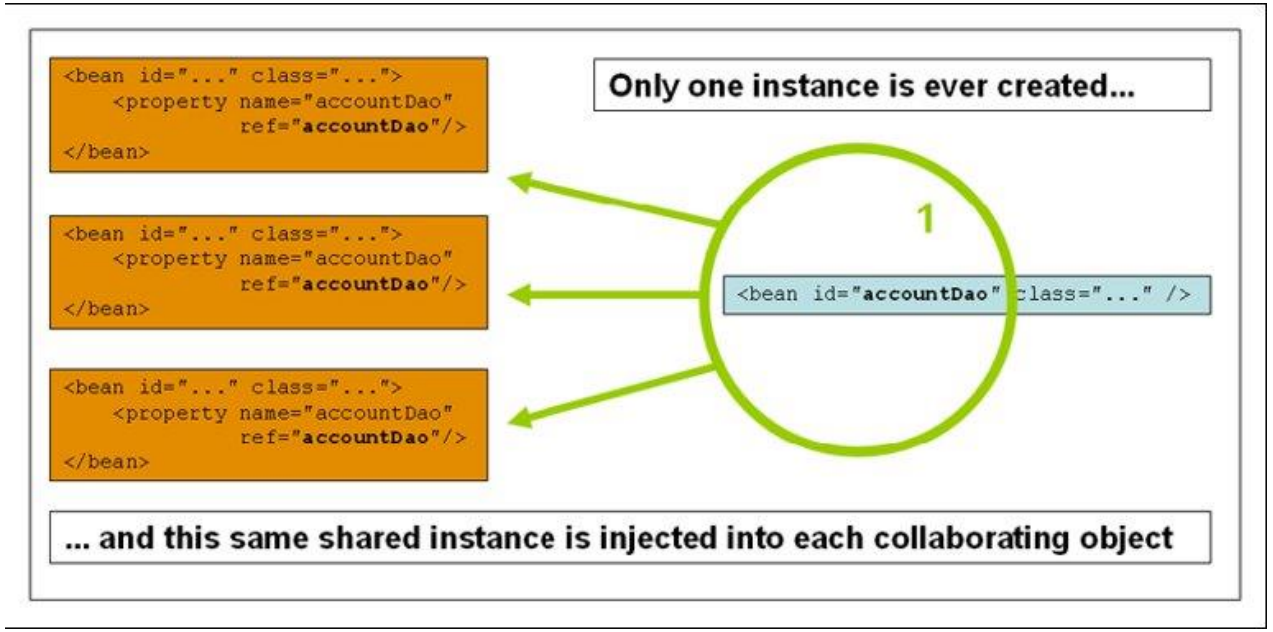
Bean scopes

- Singleton
- Prototype

Bean scopes

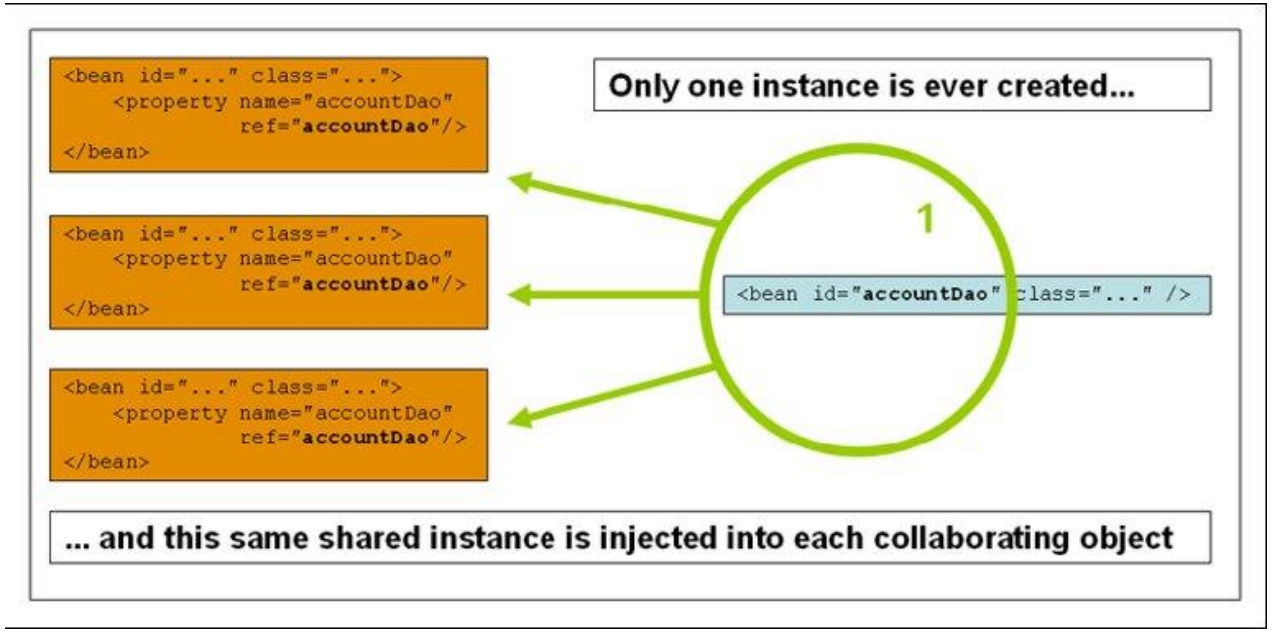
Singleton

- By default
- Single bean instance in container



Bean scopes

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



Property files with context

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

externalize

```
jdbc.url=jdbc:hsqldb:hsql://production:9002
```

```
<bean id="dataSource" class="...">
    <property name="url" value="${jdbc.url}" />
    …
</bean>
```

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" />**

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.properties:

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

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;
        }
        …
}
```

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>
```

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>
```

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;
}
```

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;
}
```

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

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 the container - otherwise UnsatisfiedDependencyException** is thrown.

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

Autowiring

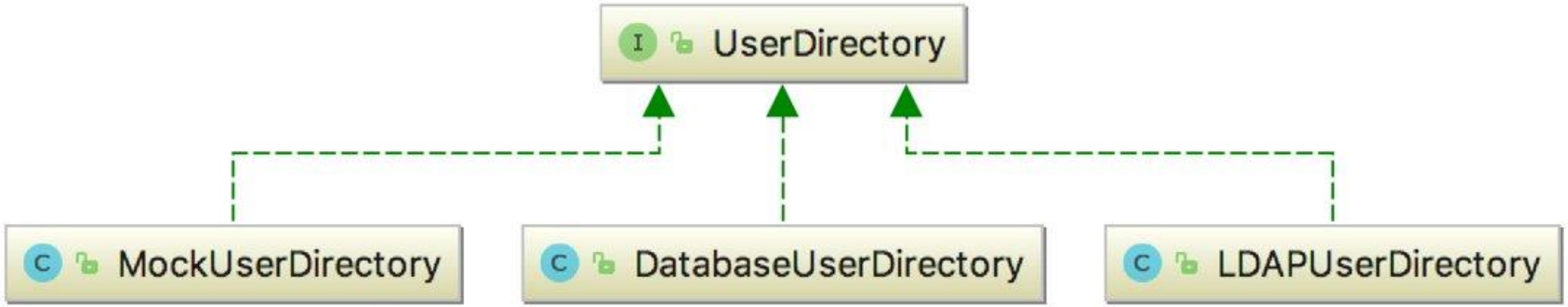
If there is more than one bean of a given type and we try to autowire byType, we are getting an error like the following:

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':

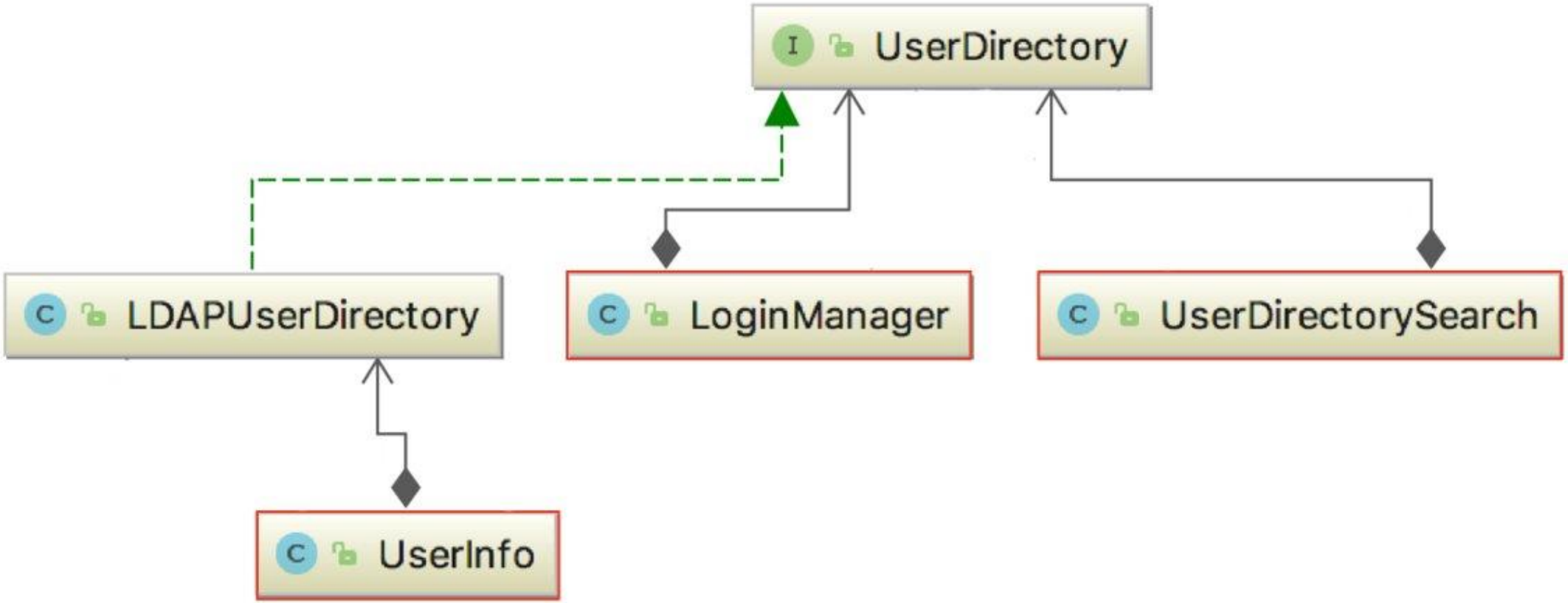
No qualifying bean of type [com.luxoft.springioc.example14.UserDirectory] is defined: **expected single matching bean but found 2: userDirectory,userDirectory2;**

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**

Autowiring


UserDirectory
MockUserDirectory
DatabaseUserDirectory
LDAPUserDirectory

Autowiring


UserDirectory
LDAPUserDirectory
LoginManager
UserDirectorySearch
UserInfo

Autowiring

Let’s have classes which need 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>
```

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" />
```

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>
```

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>
```

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>
```