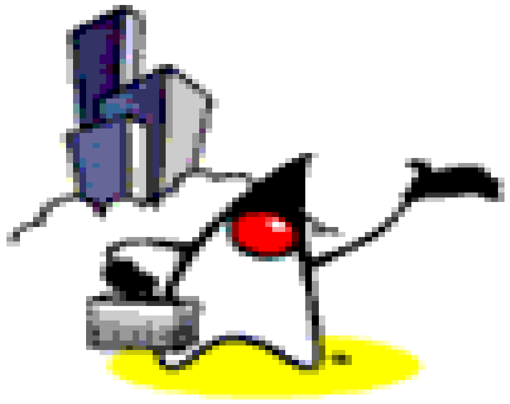
Spring Framework Dependency Injection (DI) Basics

Sang Shin
JPassion.com
“Code with Passion!”

Topics

- What is and Why Dependency Injection (DI)?
- Two DI variants
- Reading configuration
- Bean configuration
- Bean parameter types
- Auto-wiring and auto-scanning
- Bean naming



What is and Why Dependency Injection (DI)?

What is Dependency Injection (DI)?

- Also know as Inversion of Control (IoC)
- “Hollywood Principle”
  - Don't call me, I'll call you (“DI Container” is the agent)
- “DI Container” resolves dependencies of components by wiring/injecting dependency objects (push)
  - As opposed to a component looks for and instantiates dependency objects (pull)
- Termed by Martin Fowler

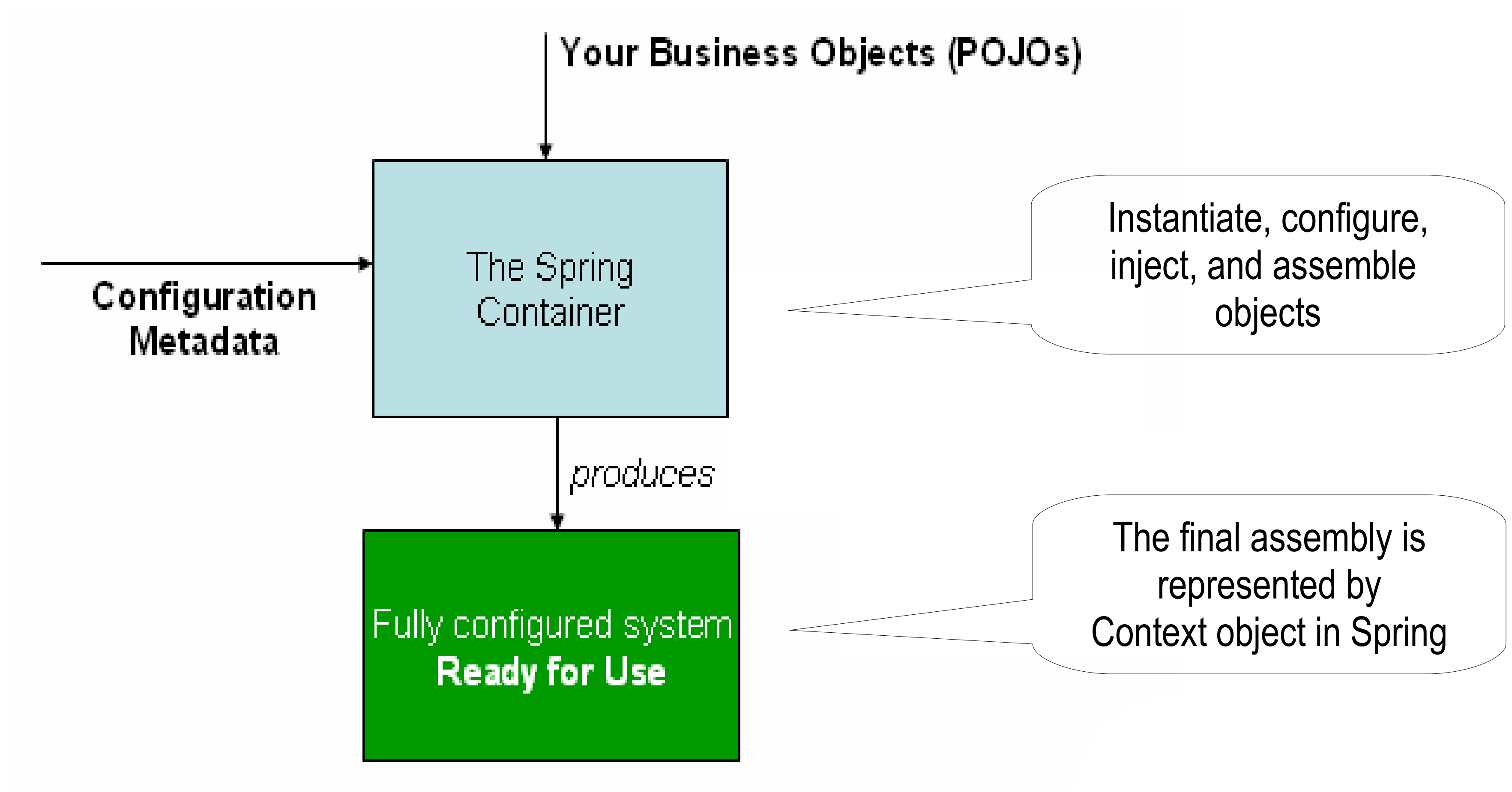
Why Dependency Injection?

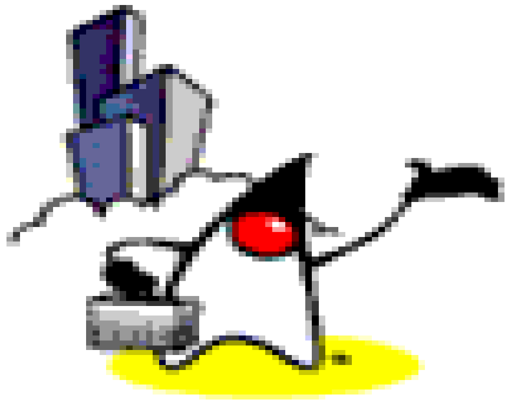
- Target components are testable as POJO's
  - During testing, dependency objects are mock objects provided by testing framework
  - During production environment, dependency objects are real objects
- Target components are more reusable and maintainable
  - No need to have a lookup code in the target component
  - Allows reuse in different application environments by changing configuration files (XML or Java) instead of code

DI Configuration

- DI container gets its instructions on what objects to instantiate, configure, inject, and assemble by reading configuration metadata
- The configuration metadata is represented in XML and/or Java configuration

Spring DI Container


Your Business Objects (POJOs)
Configuration Metadata
The Spring Container
produces
Fully configured system Ready for Use
Instantiate, configure, inject, and assemble objects
The final assembly is represented by Context object in Spring



Two Dependency Injection Variants

Two Dependency Injection Variants

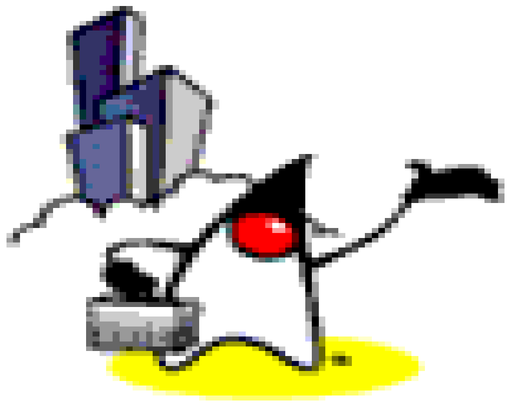
- Constructor dependency Injection
  - Dependencies are injected through constructors of a component
- Setter dependency injection
  - Dependencies are injected through setter methods of a component

Constructor Dependency Injection

```
public class ConstructorInjection {

    private Dependency dep;

    public ConstructorInjection(Dependency dep) {
        this.dep = dep;
    }
}
```

Setter Dependency Injection

```
public class SetterInjection {

    private Dependency dependency;

    public void setDependency(Dependency dependency) {
        this.dependency = dependency;
    }
}
```



Reading Configuration

DI Container Java Interfaces in Spring

- org.springframework.beans.factory.BeanFactory
  - Root interface for accessing a Spring bean container
- org.springframework.context.ApplicationContext
  - Sub-interface of BeanFactory
  - Adds easier integration with Spring's AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the WebApplicationContext for use in web applications

Reading Configuration

- #1: Reading XML configuration file
- #2: Reading Java configuration file
- #3: Reading Java configuration file with Spring Boot class

#1: Reading XML Configuration File via ClassPathXmlApplicationContext

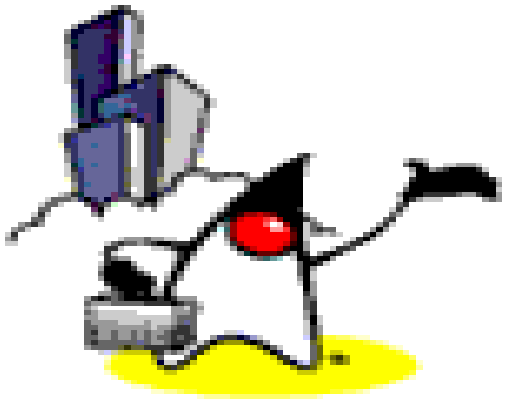
```
public class Main {

    public static void main(String[] args) {

        // Read “beans.xml” from the classpath
        ApplicationContext context =
                new ClassPathXmlApplicationContext("beans.xml");

        // Get Person object through the  factory
        Person person = (Person) context.getBean("person");
        System.out.println(person.getName());
    }

}
```

#2: Reading Java Configuration via AnnotationConfigApplicationContext class

```
@Configuration
@Import(BeanConfiguration.class)
public class MainApplication {

    public static void main(String[] args) {

        ApplicationContext context
                = new AnnotationConfigApplicationContext(MainApplication.class);

        Person person = context.getBean(Person.class);
        System.out.println(person.getName());
    }
}
```

#3: Reading Java Configuration via SpringApplication class

```
 import org.springframework.boot.SpringApplication;

@Configuration
@Import(BeanConfiguration.class)
public class MainApplication {

    public static void main(String[] args) {

        ApplicationContext context
                 = SpringApplication.run(MainApplication.class, args);

        Person person = context.getBean(Person.class);
        System.out.println(person.getName());
    }
}
```



Bean Configuration

Beans

- The term “bean” is used to refer any component managed by the *BeanFactory/ApplicationContext*
- The “beans” are in the form of JavaBeans
  - No arg constructor
  - Getter and setter methods for the properties
- Properties of beans may be simple values or more likely references to other beans
- Beans can have multiple names

Bean Configuration File (in XML)

- Each bean is defined using <*bean*> tag under the root of the <beans> tag
- The *id* attribute is used to give the bean its default name
- The *class* attribute specifies the type of the bean (class of the bean)

Bean Configuration XML File Example: Setter DI

```
<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">

                target component: should provide setter method

   <bean id="renderer" class="StandardOutMessageRenderer">
      <property name="messageProvider" ref=”provider”/
   </bean>
   <bean id="provider" class="HelloWorldMessageProvider"/>

                    dependency
</beans>
```

Target component must provide setter method

```
public class StandardOutMessageRenderer implements
    MessageRenderer {

    private MessageProvider messageProvider = null;


    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;
    }

    public MessageProvider getMessageProvider() {
        return this.messageProvider;
    }
}
```

Bean Configuration XML File Example: Constructor DI

```
<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">
```

target component: should provide constructor method

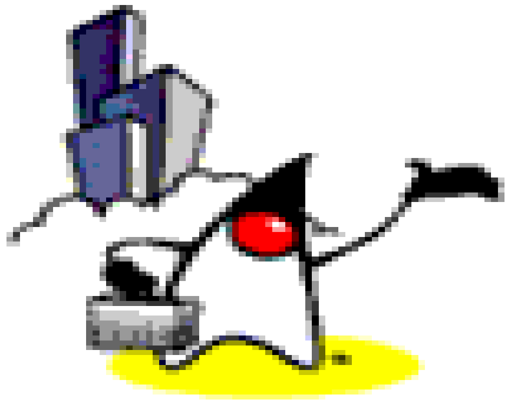
```
    <bean id="renderer" class="StandardOutMessageRenderer">
        <constructor-arg ref="provider"/>
    </bean>
    <bean id="provider" class="HelloWorldMessageProvider"/>
```

dependency

```
</beans>
```

Target component must provide constructor method

```
public class ConfigurableMessageProvider implements
    MessageProvider {

    private String message;

    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

}
```



DI Parameter Types

Injection Parameter Types

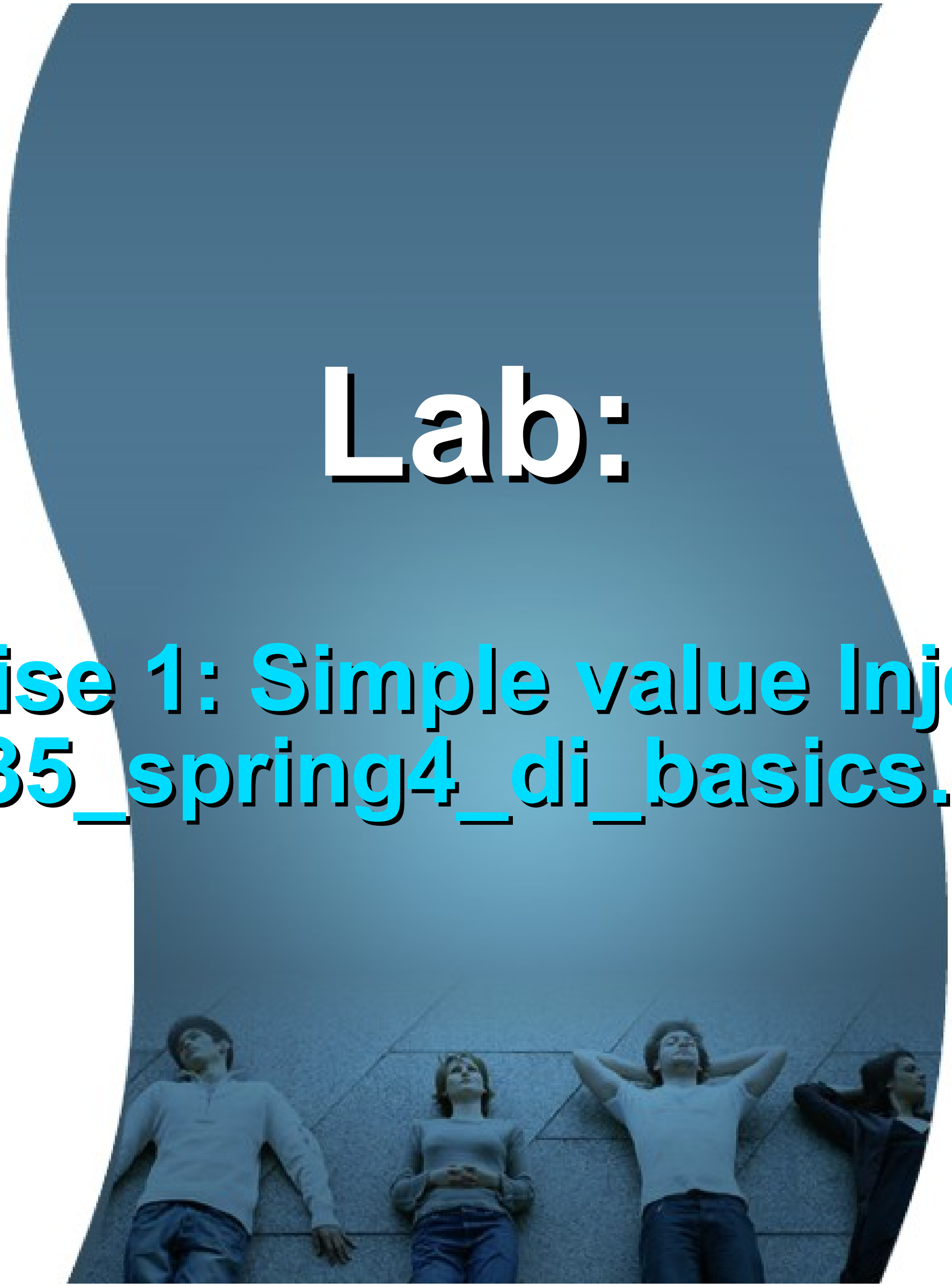
- Spring supports various kinds of injection parameters
  1. Simple values
  2. Beans
  3. Collections
- You can use these types for both setter or constructor injections

1.a Injecting Simple Values (XML)

```
<beans>

    <!-- injecting built-in values sample -->
    <bean id="injectSimple" class="InjectSimple">
        <property name="name">
            <value>John Smith</value>
        </property>
        <property name="age">
            <value>35</value>
        </property>
        <property name="height">
            <value>1.78</value>
        </property>
        <property name="isProgrammer">
            <value>true</value>
        </property>
    </bean>

</beans>
```

1.b Injecting Simple Values (Java)

```
@Configuration
public class BeanConfiguration {

    @Bean
    public Person getPerson() {
        Person person = new Person();
        person.setName("John Smith");
        person.setAge(85);
        person.setHeight(1.99F);
        person.setIsProgrammer(true);
        return person;
    }

}
```



Lab:

Exercise 1: Simple value Injection
4935_spring4_di_basics.zip

2. Injecting Beans

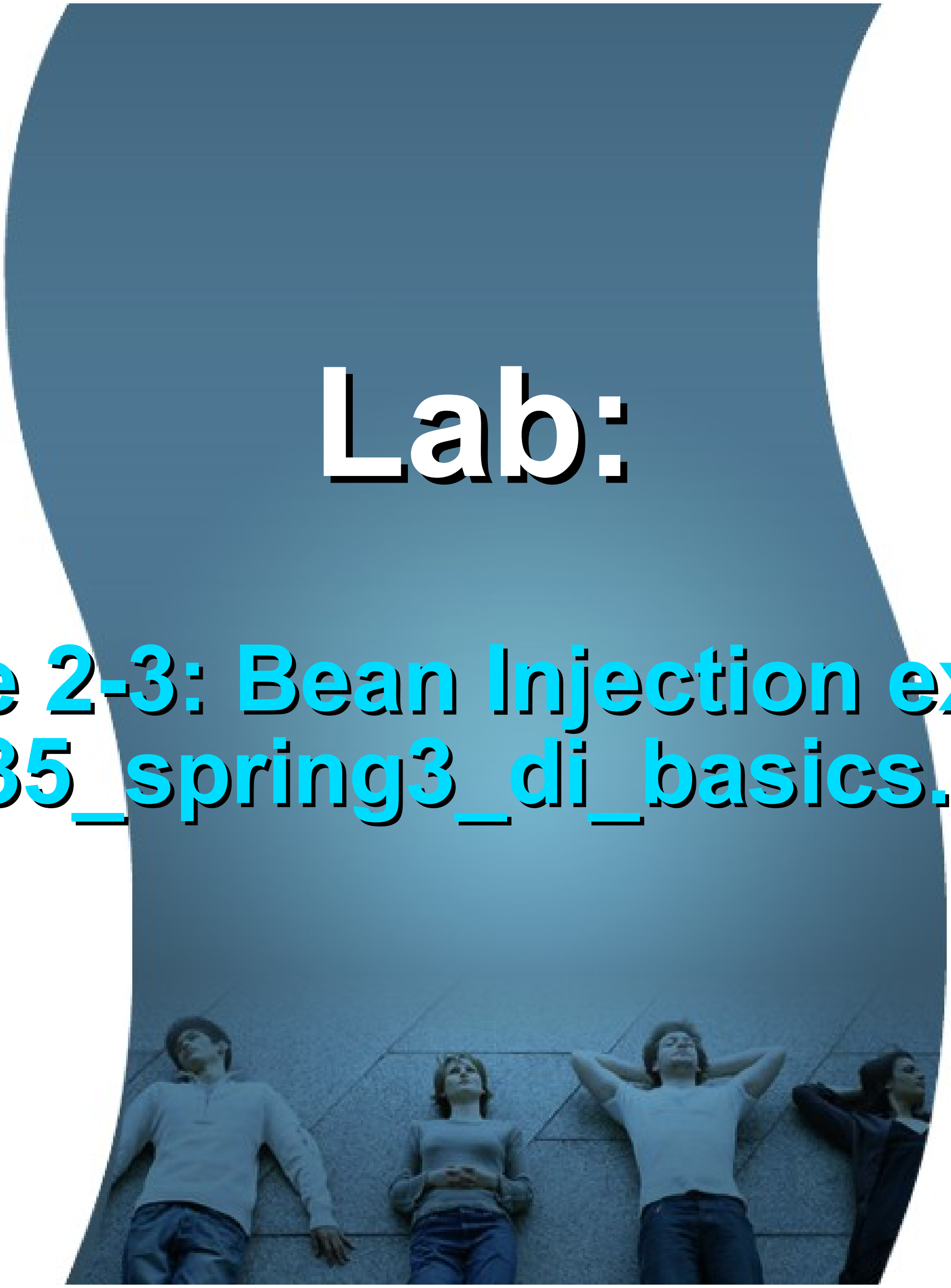
- Used when you need to inject one bean into another (target bean)
- Declare both beans first in the configuration file
- Declare an injection using <ref> tag in the target bean's <property> or <constructor-arg>

2.a Injecting Beans: Example (XML)

```
<?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">

    <!-- declare "person" bean and inject "address" bean -->
    <bean id="person" class="com.javapassion.examples.Person">
        <property name="address" ref="address"/>
    </bean>

    <!-- injected object -->
    <bean id="address"
          class="com.javapassion.examples.Address"/>

</beans>
```

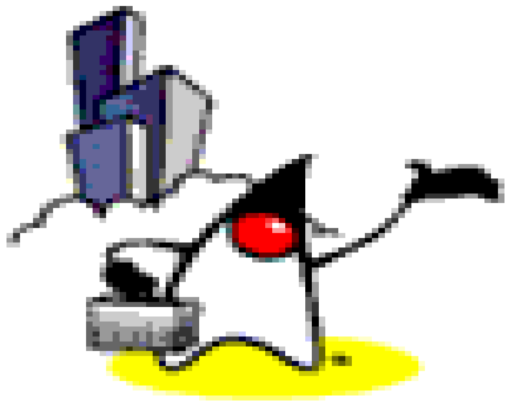
2.b Injecting Beans: Example (Java)

```
@Configuration
public class BeanConfiguration {

    @Bean
    public Address getAddress(){
        Address address = new Address();
        return address;
    }

    @Bean
    public Person getPerson() {
        Person person = new Person();
        person.setAddress(getAddress());
        return person;
    }

}
```



Lab:

Exercise 2-3: Bean Injection examples
4935_spring3_di_basics.zip



Autowiring & Autoscanning

@Autowired

- Can be used in the Java source code for specifying DI requirement (instead of in XML file)
  - For exmaple, there is no need to specify in XML

```
<property name="address" ref="address"/>
```

- Places where @*Autowired* can be used
  - Fields
  - Setter methods
  - Constructor methods
  - Arbitrary methods

Autoscan (XML)

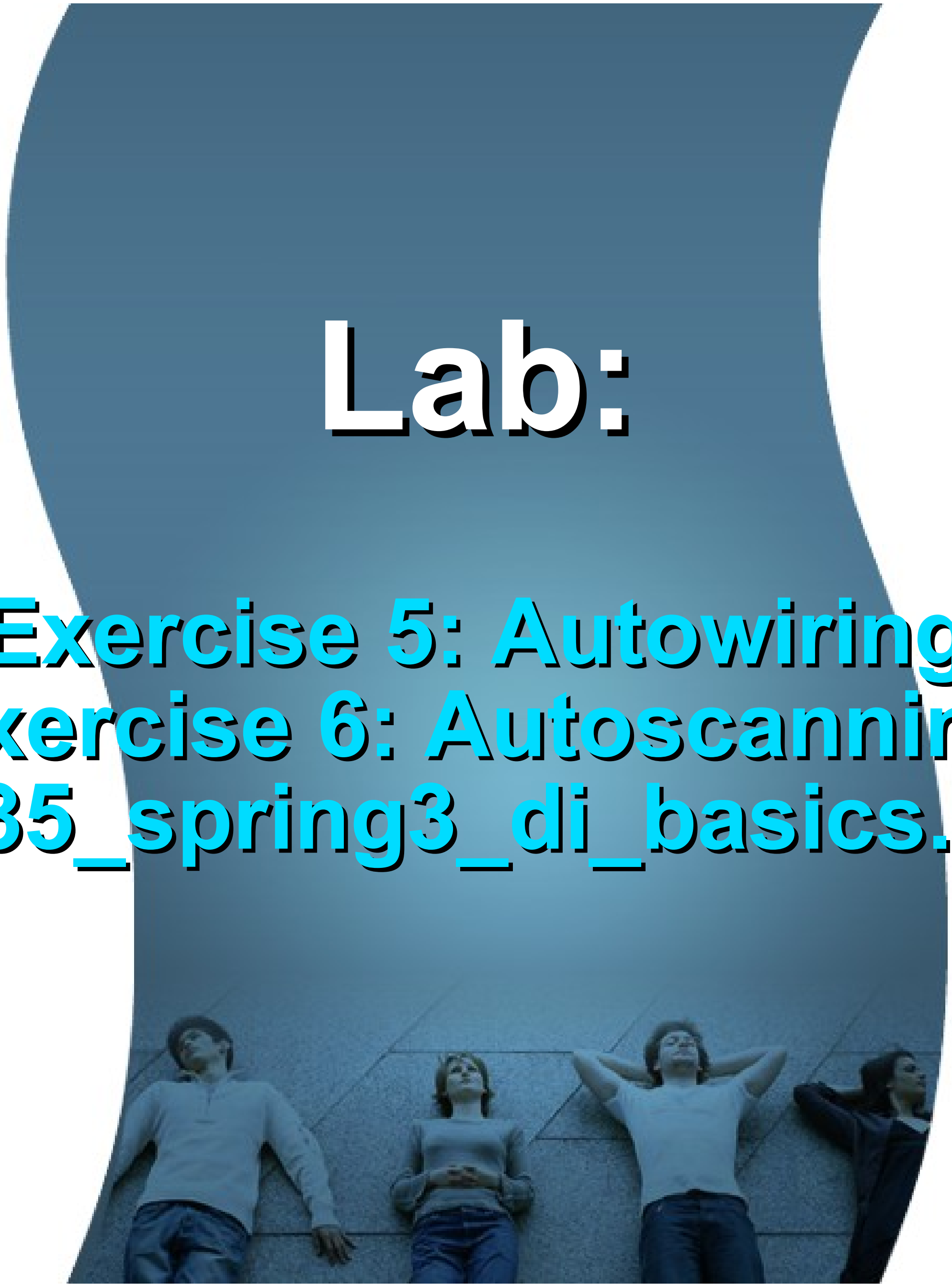
- Any bean annotated with @Component under the "com.javapassion.examples" package will be auto-detected and their instances will be created by the Spring framework

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

    <!-- enable the usage of annotations, this is optional since component-scan implies this -->
    <context:annotation-config />

    <!-- autoscan components, no need declare "address" anymore -->
    <context:component-scan base-package="com.javapassion.examples"/>

    <!-- declare "person" bean -->
    <bean id="person" class="com.javapassion.examples.Person"/>

</beans>
```

Autoscan (Java)

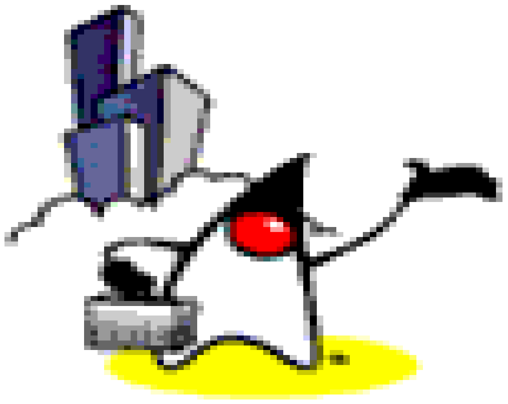
- Any bean annotated with @Component under the "com.javapassion.examples" package will be auto-detected and their instances will be created by the Spring framework

```
@Configuration
@ComponentScan
public class MainApplication {

    public static void main(String[] args) {

        ApplicationContext context = SpringApplication.run(MainApplication.class,
                args);

        Person person = context.getBean(Person.class);
        System.out.println(getPersonInfo(person));
    }

}
```



Lab:

Exercise 5: Autowiring
Exercise 6: Autoscanning
4935_spring3_di_basics.zip



Bean Naming

Bean Naming

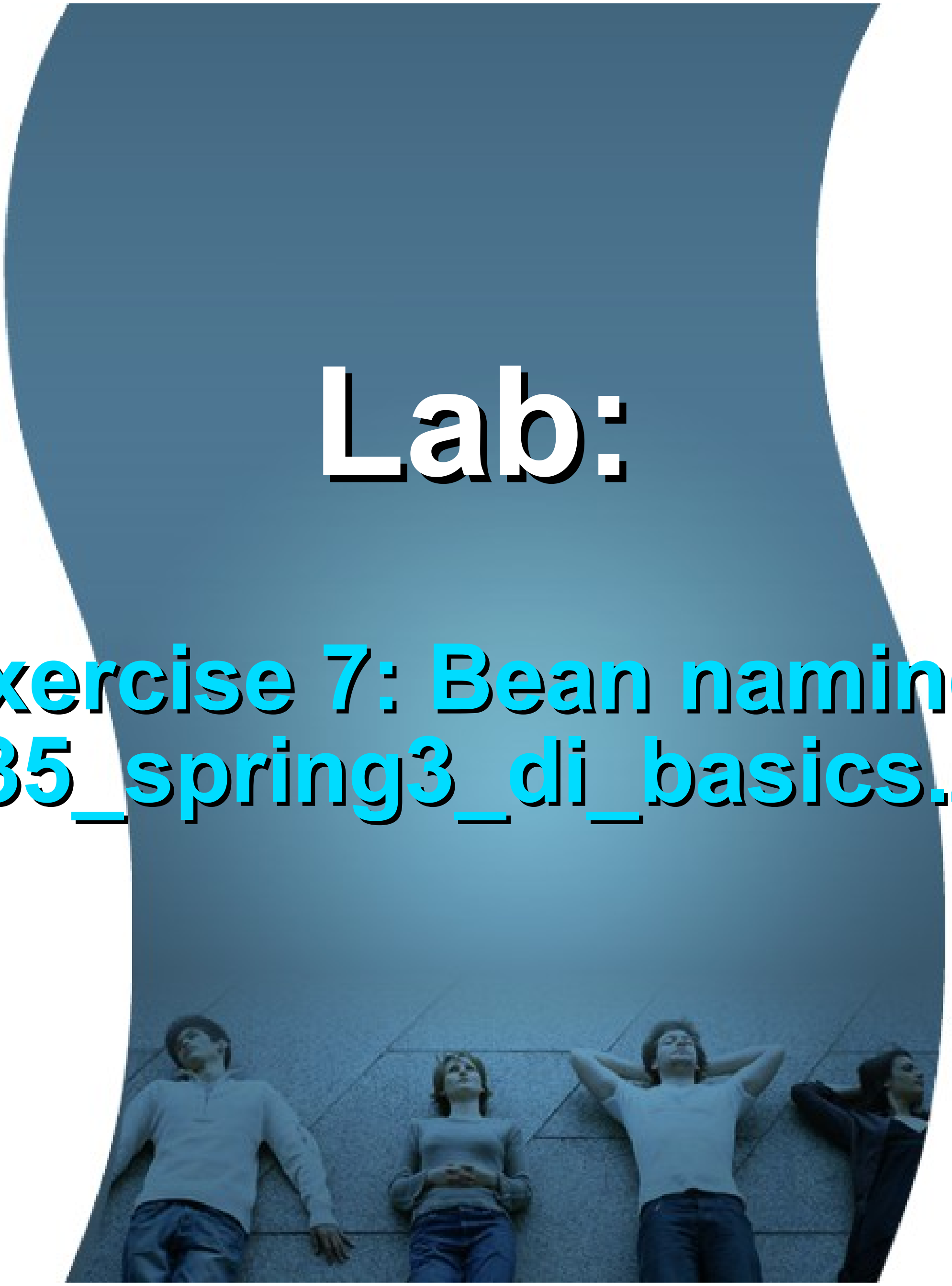
- Each bean must have at least one name that is unique within the containing BeanFactory
- Name resolution procedure
  - If a <*bean*> tag has an *id* attribute, the value of the *id* attribute is used as the name
  - If there is no *id* attribute, Spring looks for *name* attribute
  - If neither *id* nor *name* attribute are defined, Spring use the *class* name as the name
- A bean can have multiple names
  - Specify comma or semicolon-separated list of names in the name attribute

Bean Naming Example (XML)

```
<bean id=”mybeanid” class=”com.jpassion.di.Person”/>

<bean name=”mybeanname” class=”com.jpassion.di.Person”/>

<bean class=”com.jpassion.di.Person”/>

<bean id="name1" name="name2,name3,name4"
    class="com.jpassion.di.Person"/>
```

Bean Naming Example (Java)

```
@Configuration
public class BeanConfiguration {

    @Bean(name={"name1", "name2", "name3", "name4"})
    public Person getPerson() {
        Person person = new Person();
        return person;
    }

}
```



Lab:

Exercise 7: Bean naming
4935_spring3_di_basics.zip

Code with Passion!
JPassion.com