Spring Framework
Inversion of control

Part 2

LUXOFT TRAINING

LXFT LISTED NYSE

Spring Framework - Annotation-based Configuration

- Spring container may be configured with the help of annotations
- Basic supported annotations:
  - **`@Autowired`**
  - **`@Component`**
- For annotation-based configuration you should indicate in the configuration of the Spring container the following:

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

Spring Framework - Annotation-based Configuration

@Autowired
Applies to bean property setter method

```
public class SimpleMovieLister
{
    private MovieFinder movieFinder;

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

ex.20

Spring Framework - Annotation-based Configuration

`@Autowired`

- Applies to:
  - Setter methods
  - Constructors
  - Methods with any number of arguments
  - Properties (including private ones)
  - Arrays and typed collections
- Can be used with **`@Qualifier("name")`**. If this is the case, a bean with relevant ID is autowired
- By default, if there is no matching bean, an exception is thrown. This behavior can be changed with **`@Autowired(required=false)`**

Spring Framework - Annotation-based Configuration

- Only one constructor (at max) of any given bean class may carry this annotation, indicating the constructor to autowire when used as a Spring bean. This constructor does not have to be public.

- Fields are injected right after construction of a bean, before any config methods are invoked. Such a config field does not have to be public.

Task

- Change example 21 in order to make the auto-wiring of the **`movieFinder`** field and autowiring by constructor.

ex.21

Spring Framework - Annotation-based Configuration

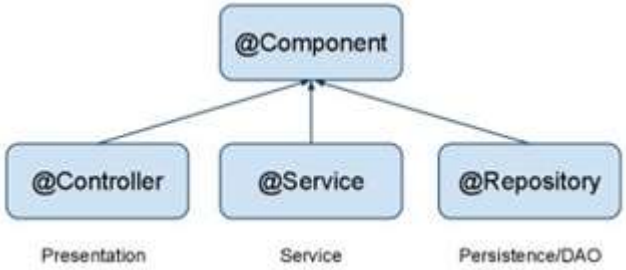
```
public class SimpleMovieLister {
   private MovieFinder movieFinder;
   @Autowired
   @Qualifier("movieFinder2")
   public void setMovieFinder(MovieFinder movieFinder) {
       this.movieFinder = movieFinder;
   }

   @Autowired
   @Qualifier("movieFinder1")
   public void prepare(MovieFinder movieFinder) {
       System.out.println("Movie finder name: " + movieFinder.getName());
   }
   …
}
```

ex.22

Spring Framework - Annotation-based Configuration

@Component

- Used for specifying Spring components without XML configuration
- Applies to classes
- Serves as a generic stereotype for every Spring-managed component
- It is recommended to use more specific stereotypes:
  - **@Service**
  - **@Repository**
  - **@Controller**



To automatically register beans through annotations, specify the following command in container configuration:

```
<context:component-scan base-package="com.luxoft.springioc.example21"/>
```

Spring Framework - Annotation-based Configuration

- You can annotate classes with **`@Component`**
- By annotating with **`@Service`**, **`@Repository`** or **`@Controller`**, the classes are more properly suited for processing by tools.
- **`@Service`**, **`@Repository`** or **`@Controller`** may carry additional semantics in future releases of the Spring Framework.

Spring Framework - Annotation-based Configuration

```
<context:component-scan base-package="com.luxoft.springioc.example23" />
```

```
@Component
public class Adder
{
    public int add(int a, int b)
    {
        return a + b;
    }
}
```

```
@Component("calculator")
public class Calculator
{
    @Autowired
    private Adder adder;

    public void makeAnOperation()
    {
        int sum = adder.add(1, 2);
        System.out.println("sum = " + sum);

    }
}
```

ex.23

Spring Framework - Annotation-based Configuration

Task

Add a multiplier to the calculator

ex.23

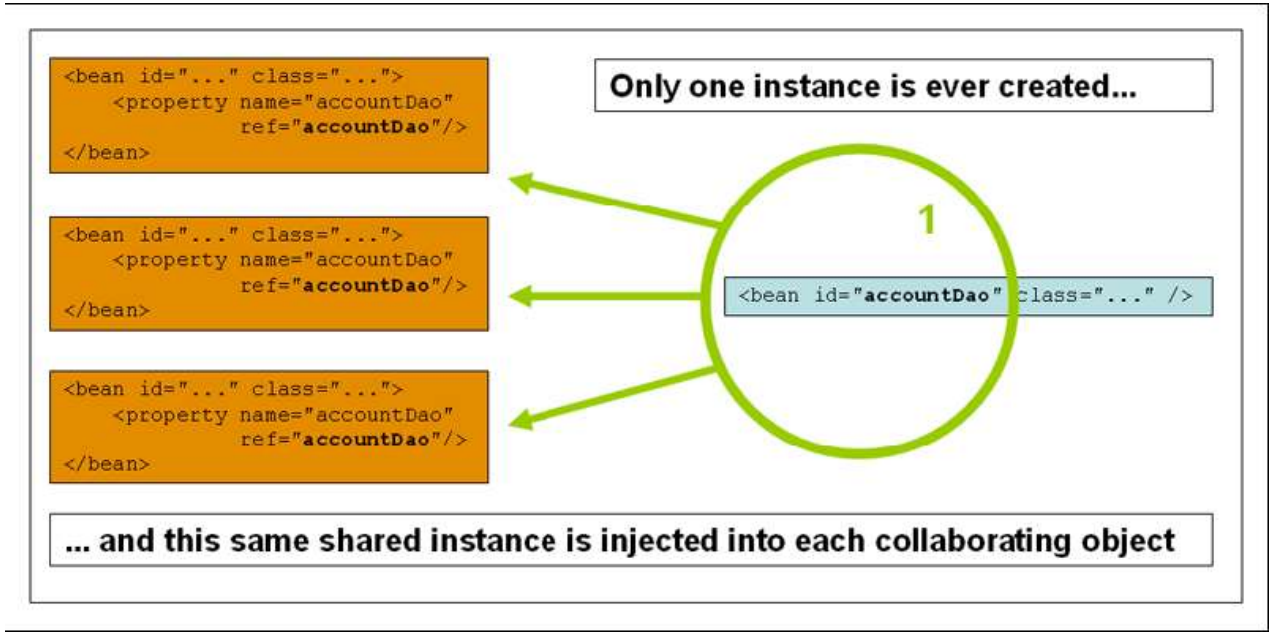
Spring Framework – Bean scopes

- General Bean Scopes:
  - Singleton
  - Prototype

- Web-specific:
  - Request
  - Session

Spring Framework – Bean scopes

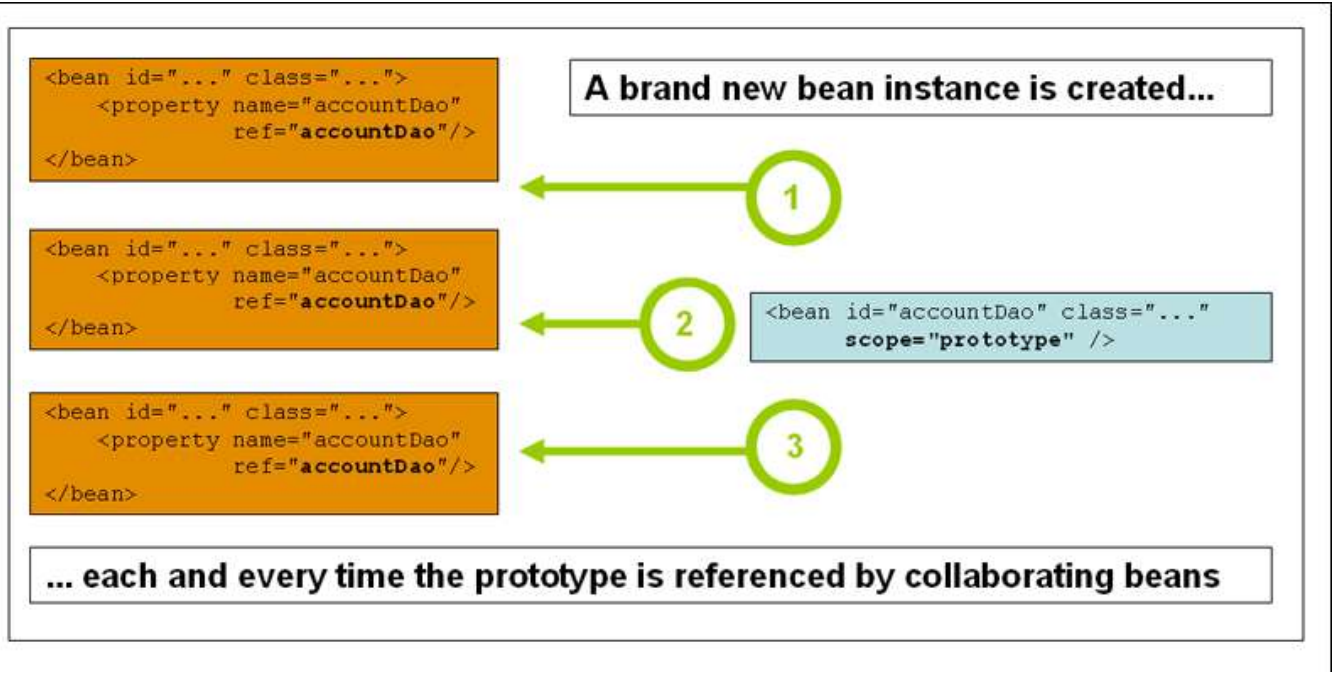
- **Singleton**
  - By default
  - Single bean instance in container



ex.24

Spring Framework – Bean Scopes

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



ex.25

Spring Framework – Prototype Scope Use Cases

- There is a class that has many configuration parameters
- You need to create instances of a class with a set of predefined configuration
- Any time you would use `new` instead of using a singleton
- You need a kind of factory method that creates instances as preconfigured in the XML

Spring Framework – Bean Scopes Using Annotations

You can also use annotation to define your bean scope:

```
@Service("bean")
@Scope("prototype")
public class Bean {
```

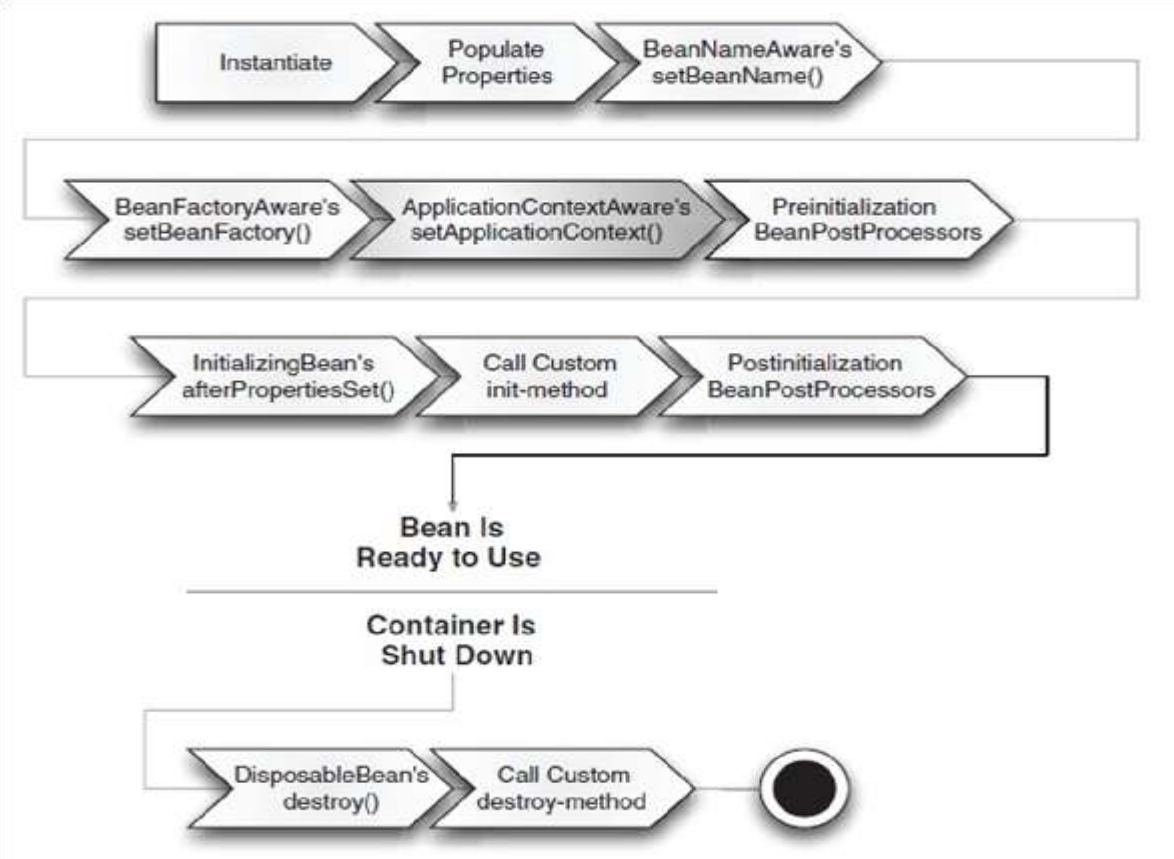
```
<context:component-scan base-package=
        "com.luxoft.springioc.example25_annotation" />
```

Task

- Change example 25_annotation in order to make the bean singleton

ex.25 annotation

Spring Framework – Bean Lifecycle



Spring Framework – Bean Lifecycle

Managing bean by implementing Spring interfaces:

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

ex.26

Spring Framework – Bean Lifecycle

Managing bean via code without dependence on Spring:

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

```
<bean id="bean1" class="Bean1" init-method="init" destroy-method="cleanup" />
```

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

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

ex.27

Spring Framework – Bean Lifecycle Demo

```
public class InitHelloWorld implements BeanPostProcessor {

   @Override
   public Object postProcessBeforeInitialization(Object bean, String beanName)
                  throws BeansException {
          System.out.println("BeforeInitialization : " + beanName);
          return bean;
   }

   @Override
   public Object postProcessAfterInitialization(Object bean, String beanName)
                  throws BeansException {
         System.out.println("AfterInitialization : " + beanName);
         return bean;
   }

}
```

ex.28

Spring Framework – Additional Features of ApplicationContext

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

```
public class CommandManager implements ApplicationContextAware  {
   private ApplicationContext applicationContext;

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

}
```

Spring Framework – Events

- Receiving of standard events:

```
public class Bean implements ApplicationListener<CustomEvent> {
   @Override
   public void onApplicationEvent(CustomEvent event) {…}
}
```

- Publishing of the custom events:

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

ex.29

Spring Framework – Events

- Event processing in **`ApplicationContext`** is provided by:
  - **`ApplicationEvent`** class
  - **`ApplicationListener`** interface
- When an event happens, all the beans which are registered in the container and implementing **`ApplicationListener`** interface are notified
- **`ApplicationEvent`** – major implementations:
  - **`ContextRefreshedEvent`** – creating and refreshing of **`ApplicationContext`**
  - **`ApplicationContext`** is ready to use **`ContextClosedEvent`** after use of **`close()`** method

ex.30

Spring Framework – Events

- **Example:** new employee registration

- **Possible event recipients:**
  - Informing security guards to do a pass
  - Guards are subscribing the event
  - Additionally, there could be other subscribers, like HR or accounts department

- **Task:** save employee into database

- **Solution:** create a class which will add employee to database and subscribe it to the event

Spring Framework – Events

```
public class EmployeeRegistrationEvent extends ApplicationEvent {

   private Employee employee;

   public Employee getEmployee() {
        return employee;
   }

   public EmployeeRegistrationEvent(Employee employee) {
        super(employee);
        this.employee = employee;
   }

}
```

Spring Framework – Events

```
public class EmployeeService implements ApplicationContextAware {

   private ApplicationContext context;

   @Override
   public void setApplicationContext(ApplicationContext context) {
       this.context = context;
   }

   public void registerEmployee(String surname, String firstName) {
      Employee employee = new Employee(surname, firstName);
      context.publishEvent(new EmployeeRegistrationEvent(employee));
   }

}
```

Spring Framework – Events

```
public class WriteEmployeeToDB implements
             ApplicationListener<EmployeeRegistrationEvent> {

  public void onApplicationEvent(EmployeeRegistrationEvent event) {
     System.out.println("Employee "
                        + event.getEmployee().getSurname() + " "
                        + event.getEmployee().getFirstName()
                        + " is saved into the database");
  }

}
```

ex.31

Task

- Add another event removeEmployee, subscribe to it in RemoveEmployeeFromDB

Spring Framework – Events

- **Advantages:**

  - There could be any number of subscribers
  - System complexity does not depend on amount and type of subscribers
  - Adding subscriber does not add the dependency: only the container knows about the subscriber

- **Disadvantages:**

  - Makes more difficult to know what happens inside the system

Annotation-driven Event Listener

A new feature of Spring 4 is the support of annotation-driven event listeners. It is possible to simply annotate a method of a managed-bean with `@EventListener` to automatically register an `ApplicationListener` (matching the signature of the method). Our example above can be rewritten as follows:

```
@Component
public class Bean {
    @EventListener
    public void blogModified(CustomEvent customEvent) {
        System.out.println("CustomEvent received through @EventListener");
    }
}
```

`@EventListener` is a core annotation that is handled transparently in a similar fashion as `@Autowired`: no extra configuration is necessary and the existing `<context:annotation-config/>` element enables full support for it.

ex.32 eventlistener

Annotation-driven Event Listener with Condition

```
public class CustomEvent {
   private boolean flag;

   public CustomEvent(Object source, boolean flag) {
       super(source);
       this.flag = flag;
   }



@EventListener(condition = "#customEvent.flag")
public void blogModified(CustomEvent customEvent) {
       System.out.println("CustomEvent received through @EventListener");
}

commandManager.getApplicationContext().publishEvent(
                                              new CustomEvent(new Object(), true));
commandManager.getApplicationContext().publishEvent(
                                              new CustomEvent(new Object(), false));
```

ex.32 eventlistener_flag

Task

Rewrite **`example 31`** so that it uses an annotation driven event listener

Spring Framework – Localization

messages_EN.properties

```
customer.name=John Jones, age : {0}, URL : {1}
```

messages_DE.properties

```
customer.name=Johann Johannis, Alter: {0}, URL : {1}
```

```
<bean id="messageSource"
   class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename">
        <value>messages</value>
    </property>
</bean>
```

Spring Framework – Localization

```
public static void main(String[] args) {
    ClassPathXmlApplicationContext context = new
        ClassPathXmlApplicationContext("example33/localization.xml");
    String nameEnglish = context.getMessage("customer.name",
                        new Object[] { 28, "http://www.luxoft.com" }, Locale.ENGLISH);
    System.out.println("Customer name (English) : " + nameEnglish);

    String nameGerman = context.getMessage("customer.name",
                        new Object[] { 28, "http://www.luxoft.com" }, Locale.GERMANY);
    System.out.println("Customer name (German) : " + nameGerman);

    context.close();

}
```

Task

ex.33

- Define one more locale and print message using it

Spring Framework – Configuration Profiles

```
<beans profile="dev">
  <bean id="bean" class="com.luxoft.springioc.example34.Bean">
      <property name="a" value="1"/>
  </bean>
</beans>

<beans profile="production">
  <bean id="bean" class="com.luxoft.springioc.example34.Bean">
      <property name="a" value="2"/>
  </bean>
</beans>
```

Spring Framework – Configuration Profiles

Setting profile and configuration loading in the Java code:

```
GenericXmlApplicationContext context = new GenericXmlApplicationContext();
context.getEnvironment().setActiveProfiles("dev");
context.load("classpath:example34/*.xml");
context.refresh();
```

Setting profile in the command line parameters:

```
-Dspring.profiles.active="profile1,profile2"
```

Task

- Define profile "testing", inject a=100, print it in Main

ex.34

Spring Framework – Java-based Configuration

```
@Configuration
public class BeanConfig {
   @Bean
   public ExampleBean bean() {
      ExampleBean exampleBean = new ExampleBean();
      exampleBean.setA(1);
      return exampleBean;
   }
}
```

```
AnnotationConfigApplicationContext context =
    new AnnotationConfigApplicationContext(BeanConfig.class);
ExampleBean exampleBean = context.getBean("bean", ExampleBean.class);
System.out.println(exampleBean.getA());
```

ex.35

Task

Define **`PersonBean`**, add this bean to **`BeanConfig`**, set name **`"Johnson"`**, print it in **`Main`**

Spring Framework – Java-based Configuration

Example from the Spring Quick Start:

```
@Configuration
@ComponentScan
public class Application {

  @Bean
  MessageService mockMessageService() {
    return new MessageService() {
      public String getMessage() {
        return "Hello World!";
      }
    };
  }
```

Spring Framework – Java-based Configuration

```
public interface MessageService {
    String getMessage();
}

@Component
public class MessagePrinter {
     @Autowired
     private MessageService service;

     public void printMessage() {
          System.out.println(service.getMessage());
     }
}
```

Spring Framework – Java-based Configuration

```
public static void main(String[] args) {
  AnnotationConfigApplicationContext context =
          new AnnotationConfigApplicationContext(Application.class);
      MessagePrinter printer = context.getBean(MessagePrinter.class);
      printer.printMessage();

      context.close();
  }
}
```

ex.36

Spring Framework – The @Import Annotation

```
public class Bean1 {
    void print() {
      System.out.println("Bean1");
    }
}


public class Bean2 {
    void print() {
      System.out.println("Bean2");
    }
}
```

Spring Framework – The @Import Annotation

```
@Configuration
public class ConfigBean1 {
    @Bean
    public Bean1 bean1() {
        return new Bean1();
    }
}

@Configuration
@Import(ConfigBean1.class)
public class ConfigBean2 {
    @Bean
    public Bean2 bean2() {
        return new Bean2();
    }
}
```

Spring Framework – The @Import Annotation

```
public class MainApp {
   public static void main(String[] args) {
      AnnotationConfigApplicationContext ctx =
            new AnnotationConfigApplicationContext(ConfigBean2.class);

      Bean1 bean1 = ctx.getBean(Bean1.class);
      Bean2 bean2 = ctx.getBean(Bean2.class);

      bean1.print();
      bean2.print();
      ctx.close();
   }
}
```

Bean1
Bean2

ex.37

Spring Framework – @ImportResource and @Value Annotation

- In applications where **`@Configuration`** classes are the primary mechanism for configuring the container, it will still likely be necessary to use at least some XML. Simply use **`@ImportResource`** and define only as much XML as is needed.

- **`@Configuration`** classes are just another bean in the container. They can take advantage of **`@Autowired`** and **`@Value`** injection.

Spring Framework – @ImportResource and @Value Annotation

```
@Configuration
@ImportResource("classpath:example38/application-context.xml")
public class AppConfig {

    @Value("${jdbc.url}")
    private String url;

    @Value("${jdbc.username}")
    private String username;

    @Value("${jdbc.password}")
    private String password;
```

ex.38

Spring Framework – @PropertySource Annotation and Environment

- The **@PropertySource** annotation provides a convenient and declarative mechanism for adding a **PropertySource** to Spring's **Environment**.

```
@Configuration
@PropertySource("classpath:jdbc.properties")
public class AppConfig {

  @Autowired
  private Environment env;

  public Environment getEnv() {
      return env;
  }
```

ex.39

Spring Framework – @Order Annotation

- **`@Order`** defines the sort order for an annotated component
- One of the new features is the ability to use **`@Order`** with autowiring. There has always been the ability to have Spring automatically collect all beans of a given type and inject them into a collection or array
- We start from this interface:

```
public interface Printer {
  void print(String text);
}
```

Spring Framework – @Order Annotation

```
@Component
@Order(Ordered.HIGHEST_PRECEDENCE)
public class ConsolePrinter implements Printer {
   @Override
   public void print(String text) {
        System.out.println("ConsolePrinter: " + text);
   }
}

@Component
@Order(Ordered.LOWEST_PRECEDENCE)
public class FilePrinter implements Printer {
   @Override
   public void print(String text) {
        System.out.println("FilePrinter: " + text);
   }
}
```

Spring Framework – @Order Annotation

```
@Service("messageService")
public class MessageService {

   @Autowired
   private List<Printer> printers;

   public void print(String text) {
       printers.forEach((printer) -> printer.print(text));
   }
```

ex.40

Comparison of Configurations

XML	Annotations	Java-based
No additional dependencies to the dependency Injection framework.	Suits when the injected components will not vary - no scenarios where different implementations of a component would be used.	Type safe: compiler will report issues if you are configuring a wrong way.
May inject mock implementations for many components, without recompiling the modules that will use them.	Less tedious to write than XML configuration.	Search and navigation is much simpler.
Flexibile when testing or running in local or production environment.	Allows mixing, e.g. dependency injection portion of the application may be XML-based, marking a method as transactional may use annotation.	Refactoring, code completion, finding references in the workspace.

Exercise

Lab guide:

- Exercise 2

Exercise

Lab guide:

- Exercise 3