it courseware™

TRAINING MATERIALS FOR IT PROFESSIONALS

Introduction to Spring 5 and JPA 2

For more information about Java Enterprise Java, or related courseware, please contact us. Our courses are available globally for license, customization and/or purchase.

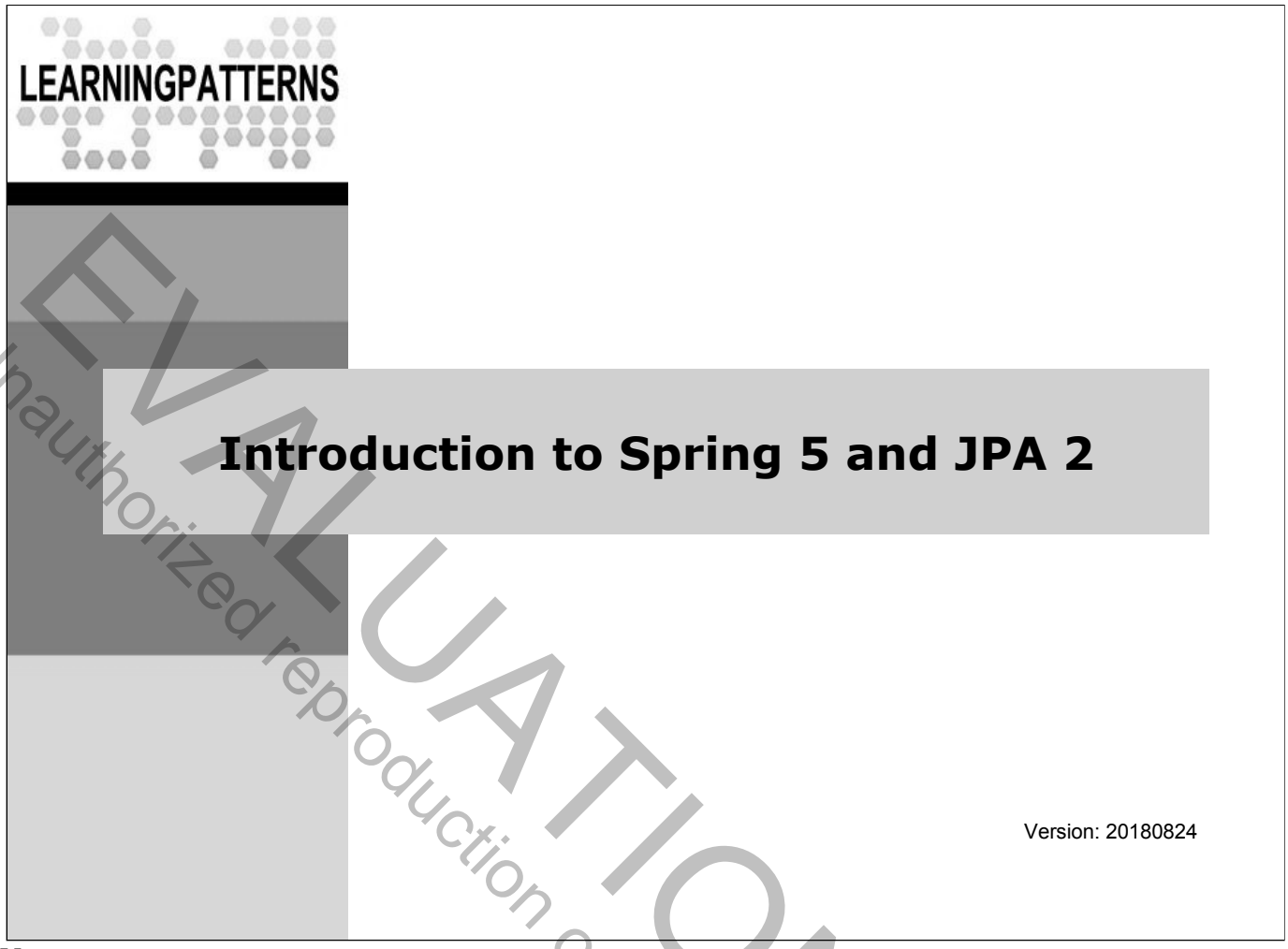
LearningPatterns. Inc. Services@learningpatterns.com | www.learningpatterns.com

Global Courseware Services 982 Main St. Ste. 4-167 | Fishkill NY, 12524 USA
212.487.9064 voice and fax

LearningPatterns

Your Source for Quality Java™ Technology Courseware

Table of Contents – Introduction to Spring 5 and JPA



Notes:

- Version: 20180824
  - Spring Base: 20180521-b
  - JPA Base: 20180820

Workshop Overview

- An in-depth course teaching the use of Spring 5 and JPA 2 to build data-driven enterprise Java applications
  - And demonstrate best practices for Spring and JPA
- The course covers the following areas of Spring technology
  - **Architecture and core features of Spring**
  - **Spring Boot**
  - **Data Access Features including Spring Data**
  - **JPA Architecture and Features**
  - **Mapping and Querying Persistent Objects with JPA**
  - **Transaction Support (and its AOP Foundation)**
  - **JPA Associations Mapping**
  - **JPA Inheritance Mapping**
  - **Integration with Web Applications**

Notes:

Workshop Objectives: Spring Capabilities

- Understand the Spring framework and use its capabilities, including:
- **Spring Core**: Dependency Injection (DI) and bean lifecycle management
  - Spring configuration and API for writing Spring programs
  - XML, Java-based, and annotation-based config
- **Spring Boot**: Easing dependency management and configuration
- **Spring Testing**: JUnit intro and Spring testing support
- **Data Access**: Data access via Spring's data support
  - DataSource support and JPA-based Repositories
  - Spring Data based repositories
- **Transactions**: Controlling transactions declaratively with Spring
- **Web**: Integrating Spring with Web applications
  - Including JPA repositories

Notes:

Workshop Objectives: JPA Capabilities

- Understand and use JPA capabilities, including:
- **Object Mapping**: Create applications using JPA to map persistent Java objects to a relational database
  - Understand and use the Entity Manager
  - Understand the lifecycle of managed entities
- **Query**: Create and execute JPA queries using JPQL
- **Associations**: Map collections and associations using JPA
- **Inheritance**: Model inheritance with JPA

Notes:

Workshop Agenda

- Session 1: **Introduction to Spring**
- Session 2: **Configuration in Depth**
- Session 3: **Intro to Spring Boot**
- Session 4: **Spring Testing**
- Session 5: **Intro to the JPA**
- Session 6: **Spring / JPA Integration**
- Session 7: **JPA Inserts and Queries**
- Session 8: **Transactions**
- Session 9: **The JPA Persistence Lifecycle**
- Session 10: **JPA Entity Relationships**
- Session 11: **Spring / Web Integration (Regular Web Apps)**
- Session 12: **Spring Data Introduction**
- [Optional] Session 13: **Additional Features**

Notes:

Typographic Conventions

- Code in the text uses a fixed-width code font, e.g.:

  `Catalog catalog = new CatalogImpl()`

  – Code fragments are the same, e.g. `catalog.speakTruth()`

  – We **bold/color** text for emphasis

  – Filenames are in italics, e.g. *Catalog.java*

  – Notes are indicated with a superscript number [1] or a **star ***

  – Longer code examples appear in a separate code box - e.g.

```
package com.javatunes.teach;

public class CatalogImpl implements Catalog {
   public void speakTruth() {
      System.out.println("BeanFactories are way cool");
   }
}
```

Notes:

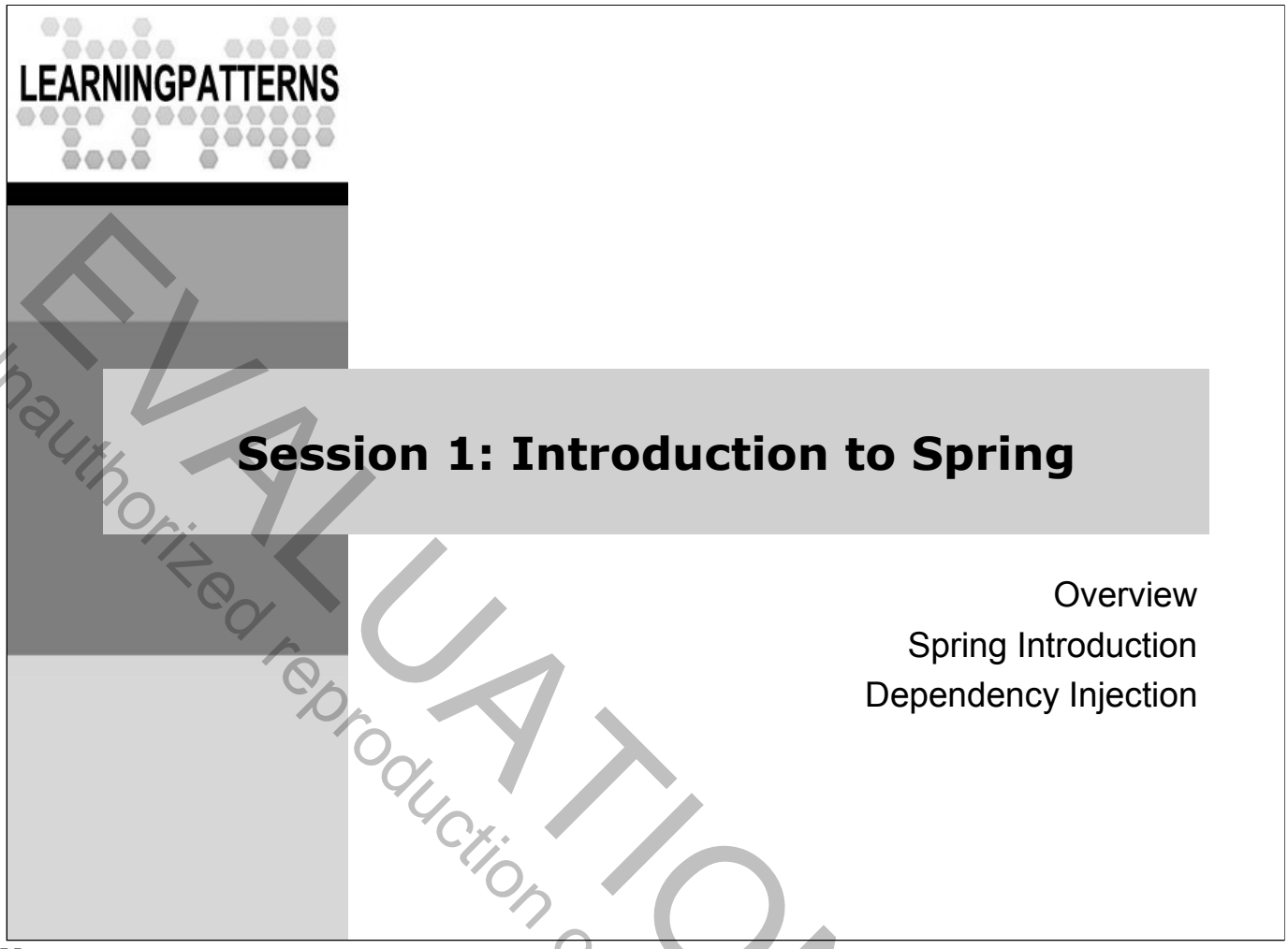
[1] If we had additional information about a particular item in the slide, it would appear here in the notes.

- We might also put related information that generally pertains to the material covered in the slide.

Labs

- The workshop has numerous hands-on lab exercises, structured as a series of brief labs
  - Many follow a common fictional case study called **JavaTunes**
    - An online music store
  - The lab details are separate from the main manual pages
- Setup zip files are provided with skeleton code for the labs
  - Students add code focused on the topic they're working with
  - There is a solution zip with completed lab code
- Lab slides have an icon like in the upper right corner of this slide
  - The end of a lab is marked with a stop like this one:

Notes:

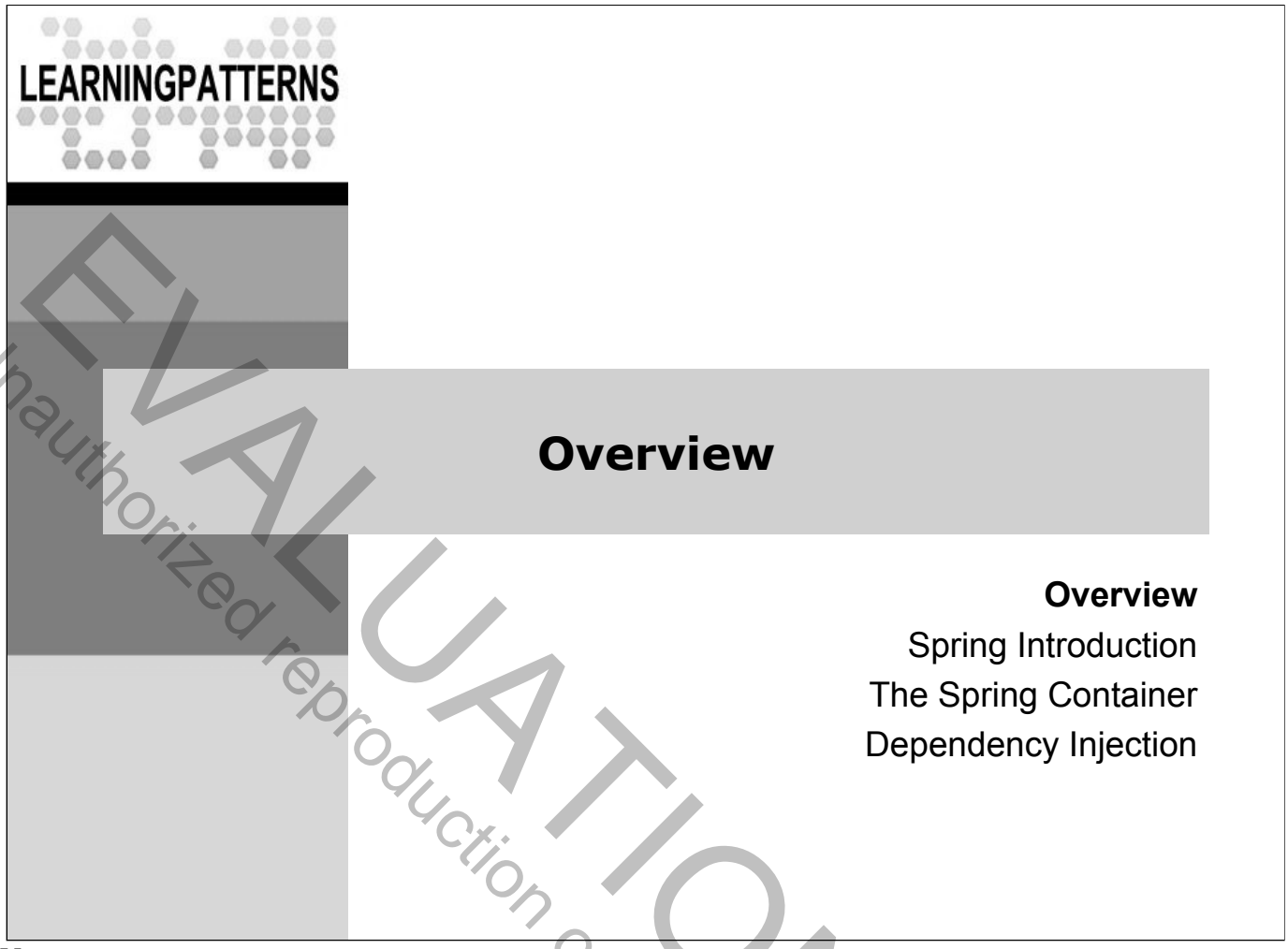


Notes:

Lesson Objectives

- Understand why we need the Spring Framework
- Understand what Spring does, and how it simplifies enterprise application development
- Learn how Spring uses configuration information and Dependency Injection (DI)
  - To manage the beans (objects) in an application
  - To manage bean dependencies

Notes:



Notes:

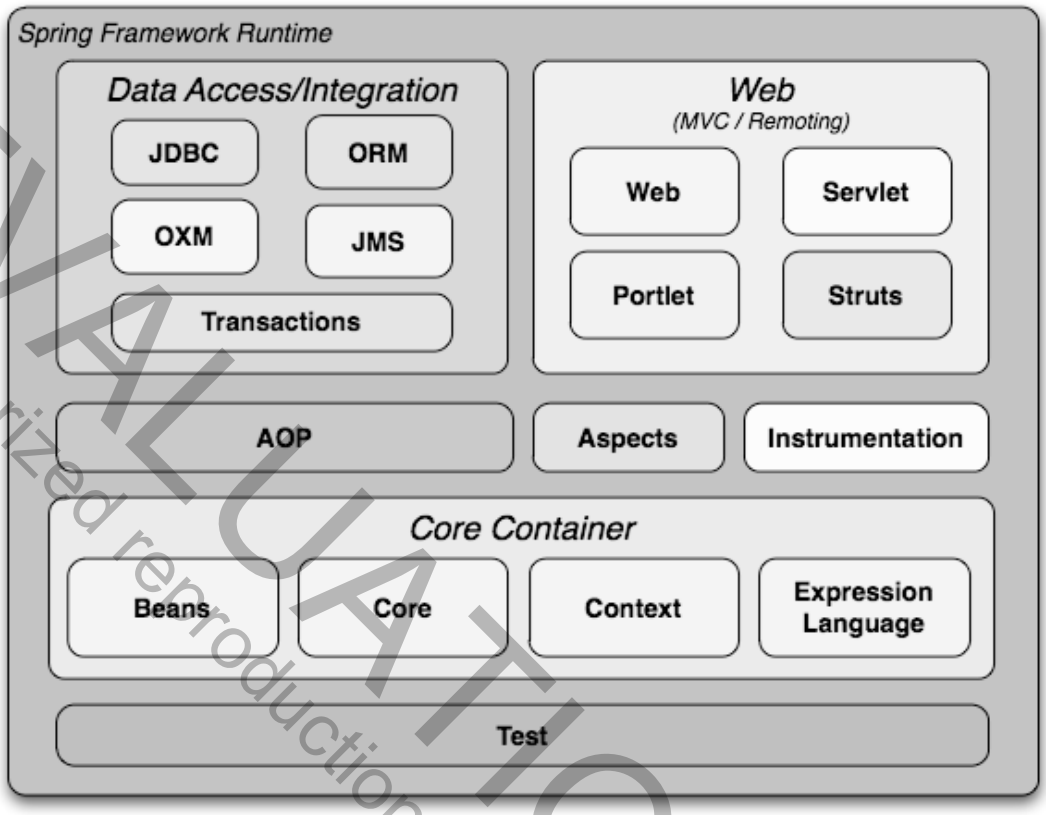
Spring and Enterprise Applications

- Enterprise apps have complex requirements, including
  - Many **application types** and dependencies
  - **Persistent data** and **transactions**
  - **Remote clients** (REST, Web Service, others)
- **Spring: Lightweight framework to build enterprise apps**
  - Non-intrusive, use only what you need, supports advanced capabilities
- Capabilities include:
  - **Dependency Injection** (**Inversion of Control/IOC**) to manage bean dependencies
  - **DAO/Repository/Transaction** support for persistent data
  - **ORM** support (Object-relational mapping, e.g. JPA)
  - **AOP**: Aspect-oriented programming
  - **Web**: Integration with standard JEE Web apps
  - **Spring MVC**: Model-View-Controller Web framework
  - **Security**: Authentication and authorization

Notes:

- There really is no formal definition of an enterprise application.
- Typically though, some of the characteristics they have are:
  - Used in a business environment, often in business-critical domains.
  - Have some form of persistent storage.
  - Have some form of remote access (Web/HTTP, Web service, Distributed Objects, etc).
  - Require some measure of scalability and fault tolerance.
- The definition of enterprise application is not important.
  - Many Java applications share some of the requirements that we are outlining here.
- DAO: Data Access Object.
- The Spring ecosystem is now very large.
  - There are other capabilities that we don't list and won't cover in this course.
  - We'll cover some of the more central technologies that many of the others rely on.

The Spring Modules



Notes:

- Module diagram from the Spring Reference Documentation

The Spring Distribution

- Spring home page: **http://spring.io/**
- Distributed as modules in separate jars
  - e.g. *spring-beans-5.0.5.RELEASE.jar*
  - Has external dependencies - e.g. logging, JUnit, etc. (1)
  - Generally use a tool like **maven** for dependencies
    - We supply jars the jars for some labs, and use maven in others
- Spring vs. JEE (Java Enterprise Edition) (2)
  - JEE similarly supports enterprise apps
    - e.g. CDI for lifecycle / dependency injection
  - Which to use? What works best
    - Based on your current and future system needs
  - You might use both - e.g. a JEE Web container
    - With Spring for lifecycle management and DI
    - Or Spring MVC layered on top of JEE Servlets/JSP

Notes:

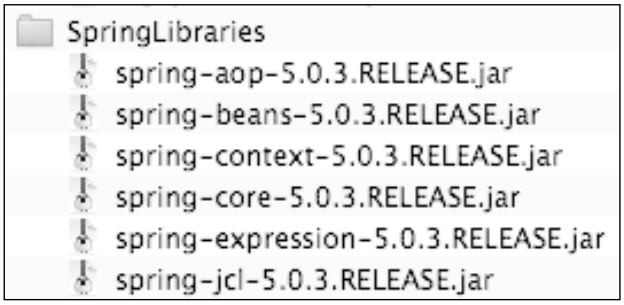
- In earlier releases (up to Spring 3), both the Spring distribution and its dependencies were available as a separate download from the Spring download pages.
  - These are no longer provided.
  - It is assumed that you'll use a build tool like maven to get the Spring libraries and its dependencies.

(1) The external dependencies for the core modules for dependency injection are (purposely) minimal.
  - Basically you just have a logging framework.
  - For other capabilities (e.g. AOP) there are more dependencies, but they're still rather small.
  - If you use other technologies however, e.g. Hibernate/JPA, then there are more external dependencies .

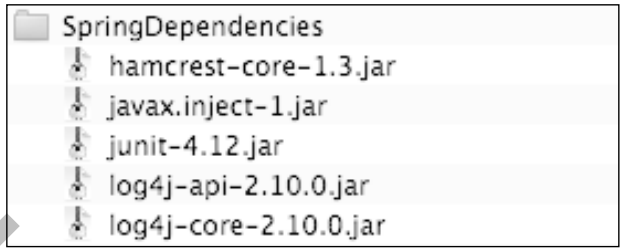
(2) Modern releases of Java/JEE offer many of the core capabilities of Spring.
  - However, it is not productive to try to compare them in an absolute sense.
  - Usually there are many constraints and requirements guiding your choice of technology.
  - We'll present Spring's capabilities, strengths, and weaknesses, to support your decision making.

The Spring jars

- At right, are the Spring libraries we supply for the early labs (1)
  - They are a subset of Spring
  - Later labs, which need more jars, use maven for dependencies



- These are external dependencies (2)
  - Again, just a subset of what we'll need in later labs



Notes:

(1) We supply the Spring project jars needed for the early labs in the lab setup.

- We downloaded these using maven and a *pom.xml* specifying the different modules available in Spring (e.g. spring-context) then gathered the jars together for the lab setup.
- The dependencies were done in the same way.
- Note that the actual jars in the lab setup may vary from this illustration - look at the setup to see what is actually used.
- Note that the junit and hamcrest jars are not needed by Spring, but needed because we use JUnit in our labs.

(2) You can see that there are not a large number of external dependencies for the basic Spring functionality.

- You'll see when we do later, more complex, labs, that maven will include/download quite a few additional dependencies.

A Word About JUnit

- **JUnit**: Open source Java testing framework
  - Often used in examples and labs to test our work
  - Labs also create console output - that's not standard (see notes)
- JUnit capabilities include:
  - **Annotations** for declaring test methods (e.g. **`@Test`**)
  - **Assertions** for testing expected results
  - **Test fixtures** for sharing common data
  - **Test runners** for running tests
- See next slide for an example
  - We'll review in more depth in the Testing session
- Most development environments have JUnit support
  - We'll use them to run tests which drive the lab code
  - The tests are the `@Test` annotated methods (see next slide)

Notes:

- We are using JUnit to drive our slide examples and lab code because it's convenient.
  - And also because it gives an easy way to test lab results.
  - However, since this is a class for learning new things, we often want some console indication of what's going on, so we'll produce some console output in our test classes.
  - That's not usually done in test classes written explicitly for testing, but it's good for our purposes.
  - We don't need to adhere to any particular testing practices - our test cases are lab-oriented, not for system testing.
  - Where appropriate, we do adhere to standard conventions - e.g. in naming our test classes and test methods.

JUnit Example

- To write a JUnit test, we:
  - Create a class, one or more methods annotated with **`@Test`**
  - Make **assertions** using static methods in **`org.junit.Assert`** (e.g. **`assertTrue`**)
- Note the **`@Test`** on `testContextNotNullPositive()`
  - The test creates the application context, and checks that it's non-null
  - We use `org.junit.Assert.`**`assertNotNull`** to perform the test
  - See notes about `import static` and `assertTrue` usage

```
// JUnit relevant code shown - some imports / code omitted
import static org.junit.Assert.*;
import org.junit.Test;

public class SpringTests {
   @Test
   public void testContextNotNullPositive() {
      ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext();
      // Just an example - we'd probably never test that the new operator works
      assertNotNull("spring container should not be null", ctx);
   }
}
```

Notes:

- The `assertXXX` methods are all static method of `Assert`.
  - The familiar way to use these methods would be to import `org.junit.Assert`, and then call the static methods through the `Assert` class.

```
Assert.assertTrue(collection.isEmpty());
```

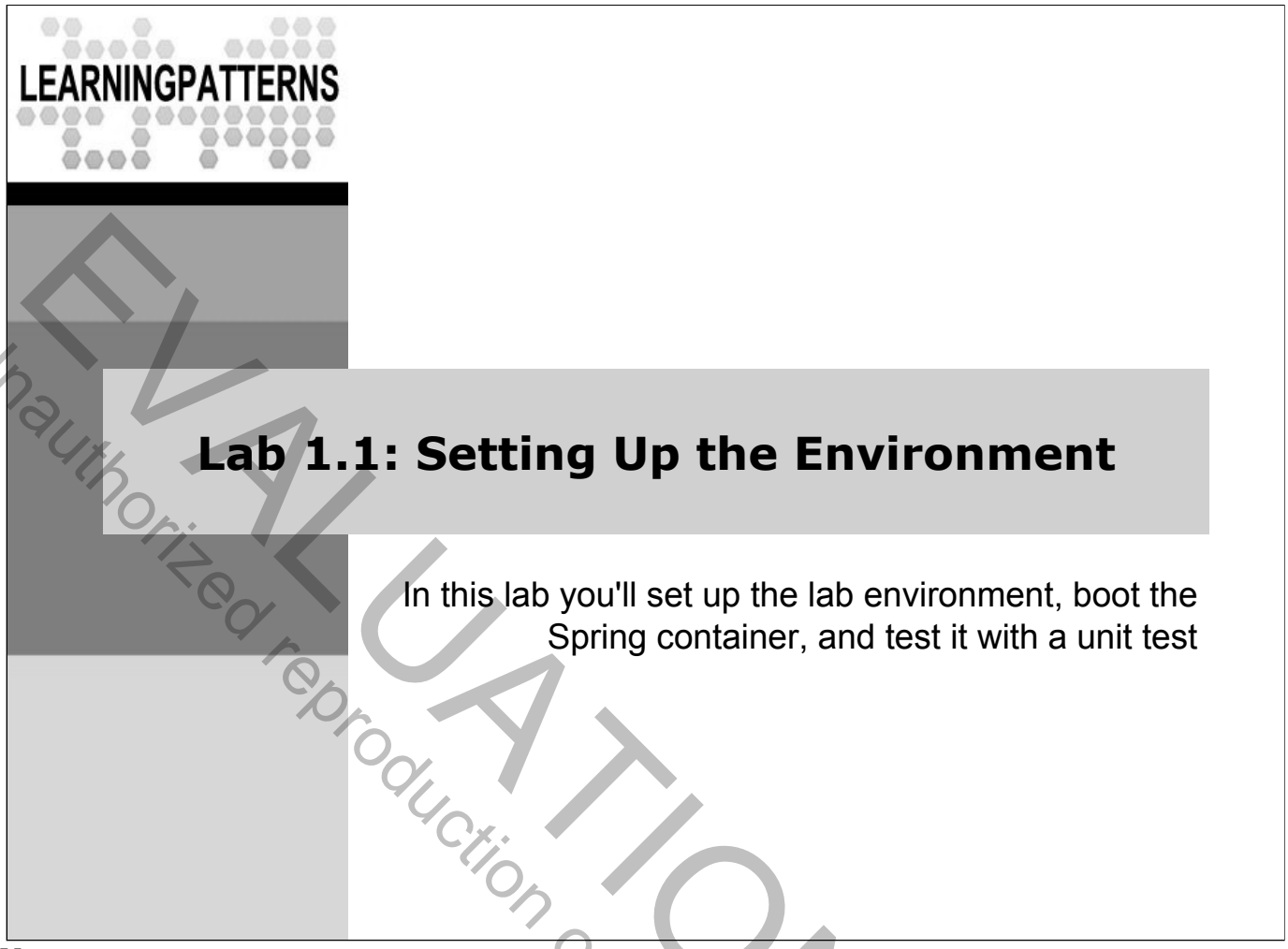
  - This is a little cumbersome, so the static import feature of Java is used - which imports static members from a class.
  - The following import statement imports all the static members (including methods) from the `Assert` class.

```
import static org.junit.Assert.*;
```

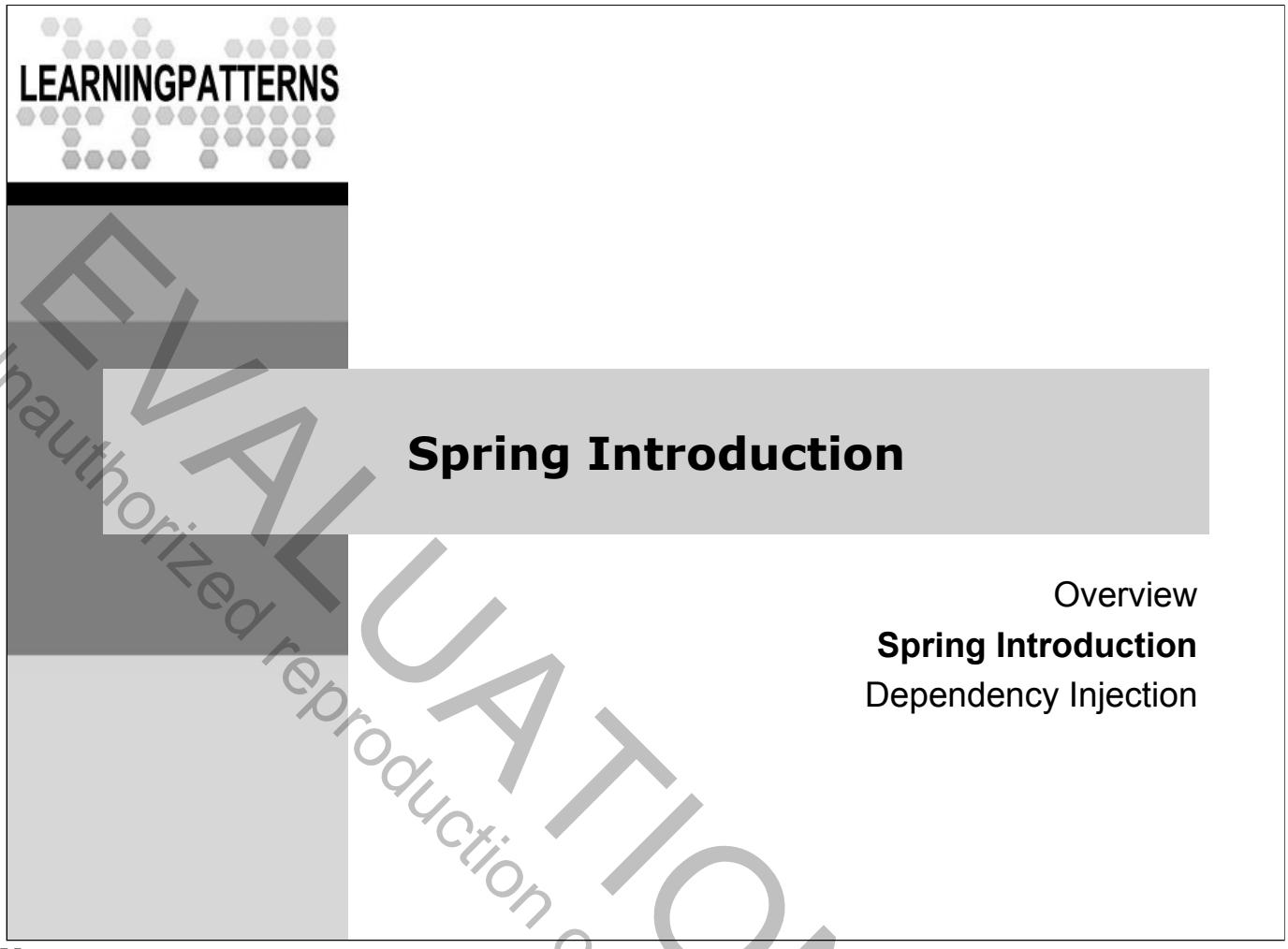
  - This allows us to use the static members without qualifying them by the classname, as seen in our earlier code.

```
assertTrue(collection.isEmpty());
```

- There is much more capability in JUnit.
  - We won't go into that, since it's beyond the scope of the course.
  - We only cover enough to show how the labs work.
  - The course is also NOT meant to teach JUnit or JUnit best practices.



Notes:



Notes:

Managing Beans: Core Spring Capability

- **Beans**: Fancy name for application objects
  - They're **POJO**s (Plain Old Java Objects)
- The **Spring container** is the "manager"
  - Uses **configuration information** (metadata) to **define, instantiate, configure, and assemble** beans
    - **Metadata**: Information about your beans (e.g. bean definitions)
  - Container uses the configuration to create and manage beans (1)
- **Configuration choices** include XML and annotations
  - **XML**: The "classic" configuration from early Spring
  - Two **annotation** choices (**`@Component`**, **`@Configuration`**)
  - We'll cover all of these

Notes:

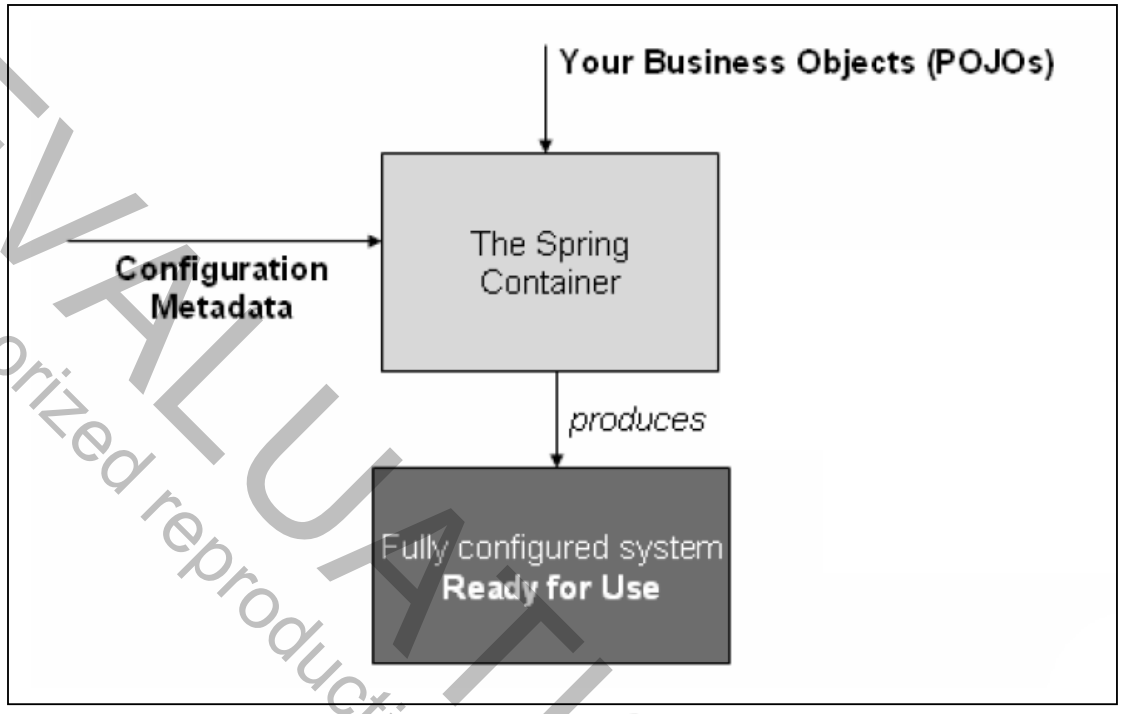
- The term bean doesn't mean that much - it's a common name (e.g. JavaBeans and EJB).
  - Here, we use it to mean an object managed by the Spring container.
- **Spring Container**: Software environment to manage beans and other capabilities.
  - Also called the Dependency Injection (DI) or Inversion of Control (IoC) container.
  - Programming to Spring basically means interacting with this software environment.

(1) Spring will create and initialize bean instances based on your configuration data.

- You can then request those instances from the Spring container by type or name.
- We'll see how this works soon.

- Configuration metadata can also be provided in the Java properties format, or even provided programmatically (using Spring's public API).
  - These are generally more cumbersome to use, and we won't cover them in this course.
  - In fact, the Spring IoC container is totally decoupled from the external form of the metadata.
  - It has its own internal format which it uses to store this information.
  - The XML format was the original one, and is still in use today, but there are now other, more sophisticated, ways to configure the container.

A Basic Spring Application



Notes:

The JavaTunes Online Store

- The course uses JavaTunes as an example and lab domain
  - A simple online music store (1)
- Some of the types you'll see include:
  - **`com.javatunes.domain.MusicItem`** : JavaBean-style value class representing a music item (e.g. an mp3)
  - **`com.javatunes.service.Catalog`** : Interface defining JavaTunes catalog functionality (including search)
  - **`com.javatunes.service.CatalogImpl`** : Concrete `Catalog` implementation (uses `ItemRepository`)
  - **`com.javatunes.persistence.ItemRepository`** : Interface defining data access API for items
  - **`com.javatunes.persistence.InMemoryItemRepository`** : Concrete `ItemRepository` implementation (simple in-memory storage)

Notes:

(1) We use a simple online music store as our domain for examples in the slides and for the labs.

- We've tried to give it enough detail to provide good material to work with while keeping it simple enough so you don't have to spend too much time in figuring it out.

Some JavaTunes Types

- `Catalog` and `CatalogImpl` are shown below
  - Note how `CatalogImpl` implements the `Catalog` interface (1)
- Let's look at how to configure some objects

```
package com.javatunes.service;

public interface Catalog {
  public MusicItem findById (long id);
  public Collection<MusicItem> findByKeyword(String keyword);
  public long size();
}
```

```
package com.javatunes.service;

public class CatalogImpl implements Catalog { // Detail omitted

  public MusicItem findById (long id) { /*  */ }
  public Collection<MusicItem> findByKeyword(String keyword)
    { /*  */ }
  public long size() { /*  */ }
}
```

Notes:

- These two types are part of JavaTunes.
  - We'll use them to motivate our discussion on how Spring works.

(1) Programming to interfaces provides many advantages by decoupling your code from concrete implementation classes.
  - This is not a concept unique to Spring.
  - Many design patterns are based on the decoupling gained by programming to an interface.
  - We'll soon see how the Spring framework makes these advantages even more usable and powerful by helping manage the dependencies that have been abstracted using interfaces.

XML Configuration Example

- Spring can be configured in an XML file
  - Default config file: ***applicationContext.xml***, but can be anything (1)
  - General structure: Top level **<beans>** containing **<bean>** elements
    - Each <bean> defines a bean
    - Generally specify **id** (a name) and **class** (fully qualified class name) (2)
  - Supports other configuration also - we'll cover it as we encounter it
- At bottom, we define one bean with XML (the metadata)
- Many existing applications use XML
  - But annotation-based approaches now recommended

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- The beans namespace is the default one for the document -->
<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.xsd">

   <bean id="musicCatalog" class="com.javatunes.service.CatalogImpl"/>

</beans>
```

Notes:

(1) *applicationContext.xml* is a standard (and default) name for the Spring config file, but you can name it anything that you want.

- In fact, as we'll see, you can have multiple config files.

(2) There are more sophisticated ways to create beans, for example with a factory class (shown later), in which case the class name may not be the actual implementation class.

- Spring provides an XML Schema for this configuration file, and the examples and lab setup files have the necessary XML namespace information in them to refer to this schema.
  - This is standard XML usage, and we don't go into the details here of how to use an XML Schema.
  - Refer to an XML reference if you need more detail on this.
- The `xsi:schemaLocation` property in the slide doesn't specify a version number for the schema.
  - The latest version will automatically be picked up.
  - For example, spring-beans-4.3.xsd (latest current version in Spring 5 as of this writing).
    - You read that correctly, the latest schema version is 4.3, even for Spring 5.
- Typically configure beans like service and repository objects, Hibernate sessions, JMS queues, etc.
- Some `BeanFactory` implementations also permit the registration of existing objects that have been created outside the factory (by user code).

The Spring Container

- The Spring container
  - Reads your configuration, and based on it:
    - **Instantiates/initializes** application objects
    - **Resolves object dependencies**
- `org.springframework.context.`**`ApplicationContext`**
  - Core API to **access the Spring container** in your code
  - Several implementations with different capabilities, e.g.
  - **`ClassPathXmlApplicationContext`**: Loads XML resources from the class path
- Interface `ApplicationContext` extends **`BeanFactory`**
  - `BeanFactory` has many core methods - usually not used directly

Notes:

- Interface `BeanFactory` is in package `org.springframework.beans.factory`.

Instantiating and Using the Container

- Below, we create a **`ClassPathXmlApplicationContext`**
  - Sometimes called "instantiating the container"
  - We pass the name of our config file (or multiple file names)
  - The container will create and manage the beans defined in the XML
- Next, we look up our configured bean via the context
  - Looked up **by type** (`Catalog.class`) - can look up **by name** also (1)
  - We do any needed work, then close the context

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

  // Code fragment - other imports and much detail omitted
  ClassPathXmlApplicationContext ctx=
    new ClassPathXmlApplicationContext("applicationContext.xml");

  // Note that getBean uses Java generics (no casting needed)
  Catalog cat = ctx.getBean(Catalog.class);
  // Or lookup by name: ctx.getBean("musicCatalog", Catalog.class);

  MusicItem item = cat.findById(1L); // Use our bean
  ctx.close(); // Close the context
```

Notes:

- Creating an instance of an ApplicationContext is often called "instantiating" or "bootstrapping" the container.
  - When the instance is created, it reads the configuration.
  - Based on the configuration, it does whatever work it needs to get ready to supply you with beans.
  - We then use that context instance to interact with the container.
  - It contains the API that you have access to.

(1) It's generally easier and more reliable to look up beans by their type.
  - However, sometimes that's not practical or possible - for example when there is more than one concrete implementation of an interface.
  - We'll see bean lookup by name later.

- We can pass multiple config files when creating the context, for example as shown below.

```
ApplicationContext ctx =
  new ClassPathXmlApplicationContext( "ctx-1.xml", "ctx-2.xml" );
```


Why Bother - What do we Gain?

- Main benefit: Our code **doesn't know about `CatalogImpl`**
  - It just knows about needed functionality (interface **`Catalog`**)
  - We've **decoupled** our code **from a dependency on the implementation** class
- Can use **any implementation** in our configuration
  - Client code **will not change**
  - That's why we **code to interfaces**, not concrete types
- Bean lifecycle is **managed by container**
  - We don't instantiate our beans directly
    - Very useful for more complex systems
  - We'll see more useful capabilities soon

Notes:

- In this simple example, it looks like we've done quite a bit of work to instantiate a single instance of a single class.
  - What is the benefit?
- The decoupling we've achieved seems like a simple thing, but it has a lot of benefits, especially when maintaining large systems.
  - We'll see how useful it can be when we explore more of the capabilities.

Summary: Working With Spring

- **Create Spring configuration data** for your beans
  - It's the "cookbook" telling Spring how to create objects
  - Via an XML file like *applicationContext.xml* or via annotations
- **Initialize the Spring container**
  - e.g. **create an application context** instance to read config data
  - It will initialize the beans in the config file(s)
- **Retrieve beans** via the context instance and use them
  - e.g. use **`getBean()`** to look up a bean by type or name
  - **Lookup by type is preferred**, unless you can't do it
    - For instance, if you have two beans implementing the same type

Notes:

- There are many, many usage scenarios for Spring.
  - And many, many different alternatives for each scenario
  - Spring is very large, and has a lot of flexibility and capability
- In this initial introduction we show you one straightforward way of using Spring.
  - We will gradually introduce more capabilities throughout the course.

More on ApplicationContext

- Access point for many Spring capabilities, including:
  - **Bean access**
  - **Resource Access**: Config files, URLs, other files
  - **Message resources with I18N** capability
  - **Publishing events** to managed beans that are listeners
  - **Multiple, hierarchical contexts**
- **`ClassPathXmlApplicationContext`**
  - Loads XML config files from the classpath
- **`FileSystemXmlApplicationContext`**
  - Loads XML config files from the file system or URLs
  - Both in `org.springframework.context.support`
- **`AnnotationConfigApplicationContext`**
  - Accepts annotated classes as input (more on this later)
  - In `org.springframework.context.annotation`

Notes:

- The `ApplicationContext` is the full-fledged representation of the Spring container.
  - It is often regarded as the type that supplies the "framework" capabilities of Spring, rather than simple bean management.
- If you are writing applications for a very resource-restricted environment, such as a mobile device, you might consider using `BeanFactory` over `ApplicationContext`.
  - Then again, even mobile devices these days usually have enough capability to make the additional resources used by `ApplicationContext` negligible.
- There are also Web-based application contexts we'll cover later.
- We'll cover the ApplicationContext API in various parts of the course.

Some BeanFactory/ApplicationContext API

- Useful methods include:
  - **`boolean containsBean(String)`**: true if named bean exists
  - **`<T> T getBean(Class<T> requiredType)`**: Get by type
  - **`<T> T getBean(String, Class<T> requiredType)`**: Get by name
  - **`Class<?> getType(String name)`**: Get type of named bean
  - **`boolean isSingleton(String)`**: Is bean a singleton
  - **`String[] getAliases(String)`**: Get any aliases for this bean
  - Many more methods - see the javadoc
- Can specify config files in multiple ways
  - ant-style wildcards: e.g. ***conf/**/ctx.xml*** - All *ctx.xml* files under *conf*
  - **`file:`** and **`classpath:`** prefixes - forces use of specified location, e.g.
    - The following loads from the classpath:
      - `new FileSystemXmlApplicationContext("classpath:ctx.xml");`
  - Spring uses its resource classes under the hood to do this

Notes:

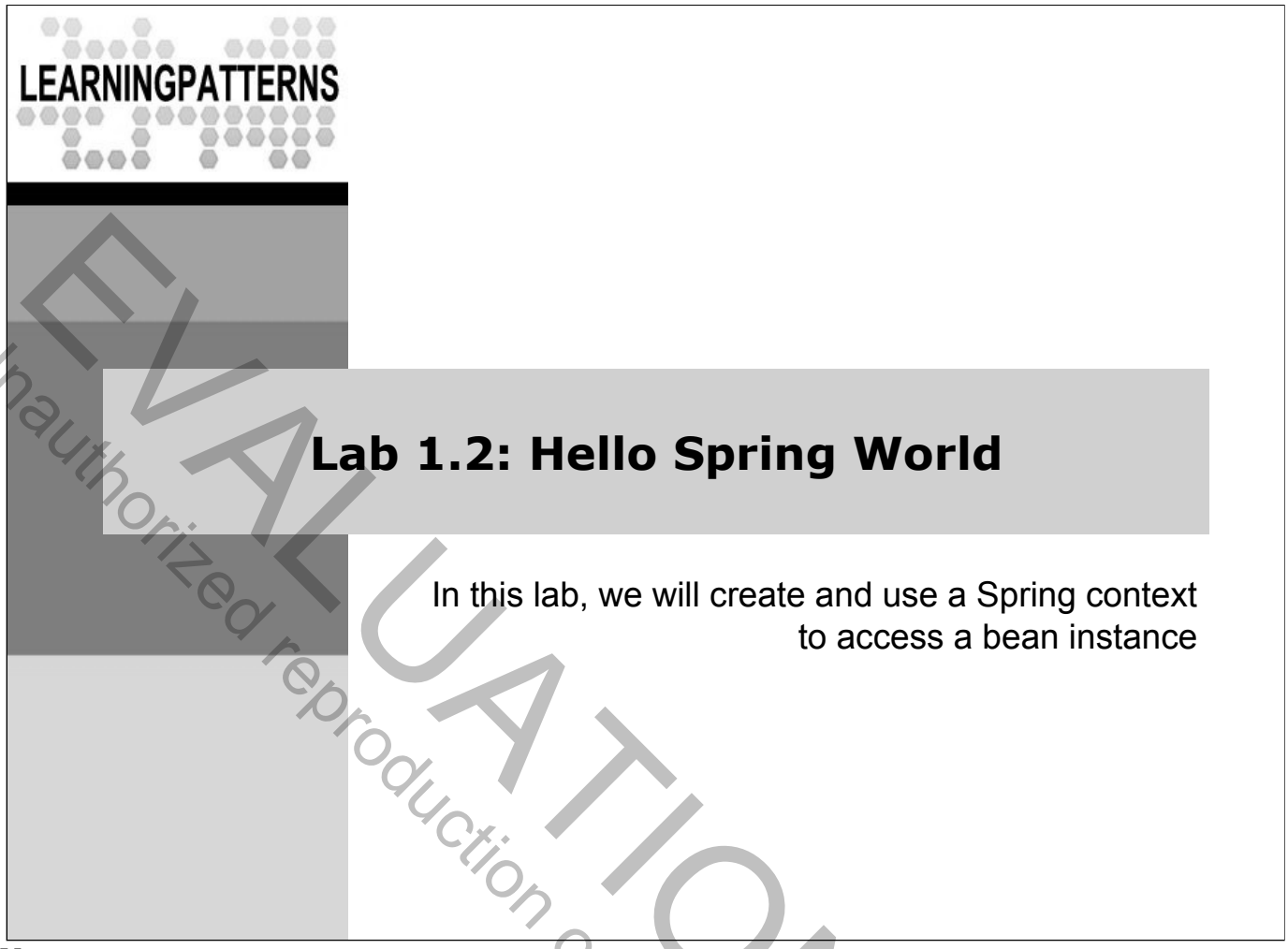
- The methods shown are a part of the `BeanFactory` API inherited by `ApplicationContext.`
- `getBean` returns either a singleton (shared) instance, or a newly created bean.
  - `NoSuchBeanDefinitionException` thrown if the bean can't be found
  - `BeansException` thrown if an exception occurred while instantiating/preparing the bean
  - `BeanNotOfRequiredTypeException` thrown if the bean is not of the required type
- `Object getBean(String)` returns an object – which is then cast to the required type.
  - The newer `getBean(String,Class<T> requiredType)` method is generally preferred.
- `getType(String)` throws `NoSuchBeanDefinitionException` if the bean can't be found.
- `isSingleton(String)` throws `NoSuchBeanDefinitionException` if bean can't be found.
- The method `<T> T getBean(String, Class<T> requiredType)` may look strange.
  - This is standard Java generics syntax - The first <T> in the return type simply indicates that this is a generic method, parameterized by the type parameter <T>.
  - The `T` return value indicates that the return type is generic (that is, it will take on different types based on the <T> parameter).
  - The `Class`<T> argument indicates that when you call the method, you pass in the class which specifies what type <T> actually is in that call.

Mini-Lab: Review Javadoc

Mini-Lab

- We provide the Spring Javadoc
  - Under *StudentWork/Spring/**Resources/SpringDocs/javadoc***
- In a browser, open *index.html* in the folder above
- Review the javadoc for the following types
  - **`BeanFactory`**
  - **`ApplicationContext`**
  - **`ClassPathXmlApplicationContext`** (especially the constructors)
  - **`FileSystemXmlApplicationContext`** (especially the constructors)

Notes:



Notes:

Annotation-Based Configuration Example

- Beans can be declared with annotations
  - **@Component** (`org.springframework.stereotype`) - Spring specific
    - `@Component` often called a "stereotype" annotation
  - **@Named** (`javax.inject`) - Standard Java (JSR 330) annotation
  - We'll use Spring style annotations in this course (1)
- Below, **@Component** declares `CatalogImpl` as a bean
  - Specifies id as **musicCatalog** (Default id: `catalogImpl`)
    - Same bean as previous XML example
  - Could also have used **@Named("musicCatalog")**

```
import org.springframework.stereotype.Component;

package com.javatunes.service;

@Component("musicCatalog") // Declares bean with id musicCatalog
public class CatalogImpl implements Catalog {
 /* Most detail omitted … */
}
```

Notes:

(1) Common usage in Spring projects is to use the Spring-based annotations.

- Some advocate the usage of JSR-330 annotations since they are a Java standard.
- This would (theoretically) allow you to port to another container that uses JSR-330 annotations - for example JEE and its CDI capabilities.
  - However, this would entail a lot more work than just the annotations on the beans, so we don't consider this much of a benefit, especially since it's relatively rare to do this kind of transition.
- In addition, the behavior of the JSR-330 annotations in the Spring container are not exactly the same as in a JEE container, so using them is somewhat misleading.
- All these considerations lead us to choose the Spring-standard annotations over the JSR-330 annotations when working with Spring.

A Brief Note on Annotations

- Standard Java mechanism to **add metadata** to source code
  - Like comments that the compiler is aware of
- **@** is used for both declaration and use of annotations (1)
  - e.g. **`@Component`**
- Annotations don't directly affect program semantics
  - They are **not executable code**
- Tools work with the annotated code
  - And may affect the semantics of a running program
  - **`@Named`**/**`@Component`** annotated beans are recognized as bean defs by the Spring container

Notes:

(1) There is a syntax for declaring annotations, a syntax for using them in declarations, a class file representation, and an API to read them.

- We'll only cover the capabilities that we'll be using for our Spring coding.

- Annotations can replace many "side files" – e.g. XML deployment descriptors.
  - It's easier for tools to read the metadata and source if they're all in one place.
  - They're often used for specifying details of high-level technologies (e.g. AOP, EJB, Web Services …).
  - These often result in code generation.

- Annotations are present in the class file, and can be read by tools.
  - The JDK also comes with a command line utility, called `apt` (Annotation Processing Tool), that can be used to process annotations.

Enabling Annotations / Detecting Beans

- Spring can automatically **scan** for annotated beans on the classpath
  - This registers them as normal beans
- Enable auto scanning via **<context:component-scan/>**
  - A standard element from Spring's `context` namespace (see next slide)
  - Includes the capabilities of `<context:annotation-config/>` (1)
  - **basePackage** attribute configures the packages to scan

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd">

  <context:component-scan base-package="com.javatunes"/>

</beans>
```

Notes:

(1) In addition to enabling scanning, `context:component-scan` also enables the capabilities of `context:annotation-config`.

- `context:annotation-config` activates the ability to detect annotations like `@Autowired` and `@Required`.
  - It only affects beans that are already registered.
  - `component-scan` element in the slide example will scan for all components in the `com.javatunes` package and all of its sub-packages, and register them as beans.
  - Note that annotations are actually detected by bean postprocessors (not important to know, but interesting if you want to know a bit about Spring internals).

- There are other attributes to component-scan, including
  - **annotation-config**: (default true): should the implicit annotation post-processors (`AutowiredAnnotationPostProcessor`, `CommonAnnotationPostProcessor`) be registered
  - **resource-pattern**: Controls the class files eligible for component scanning. Default is `**/*.class`, the recommended value

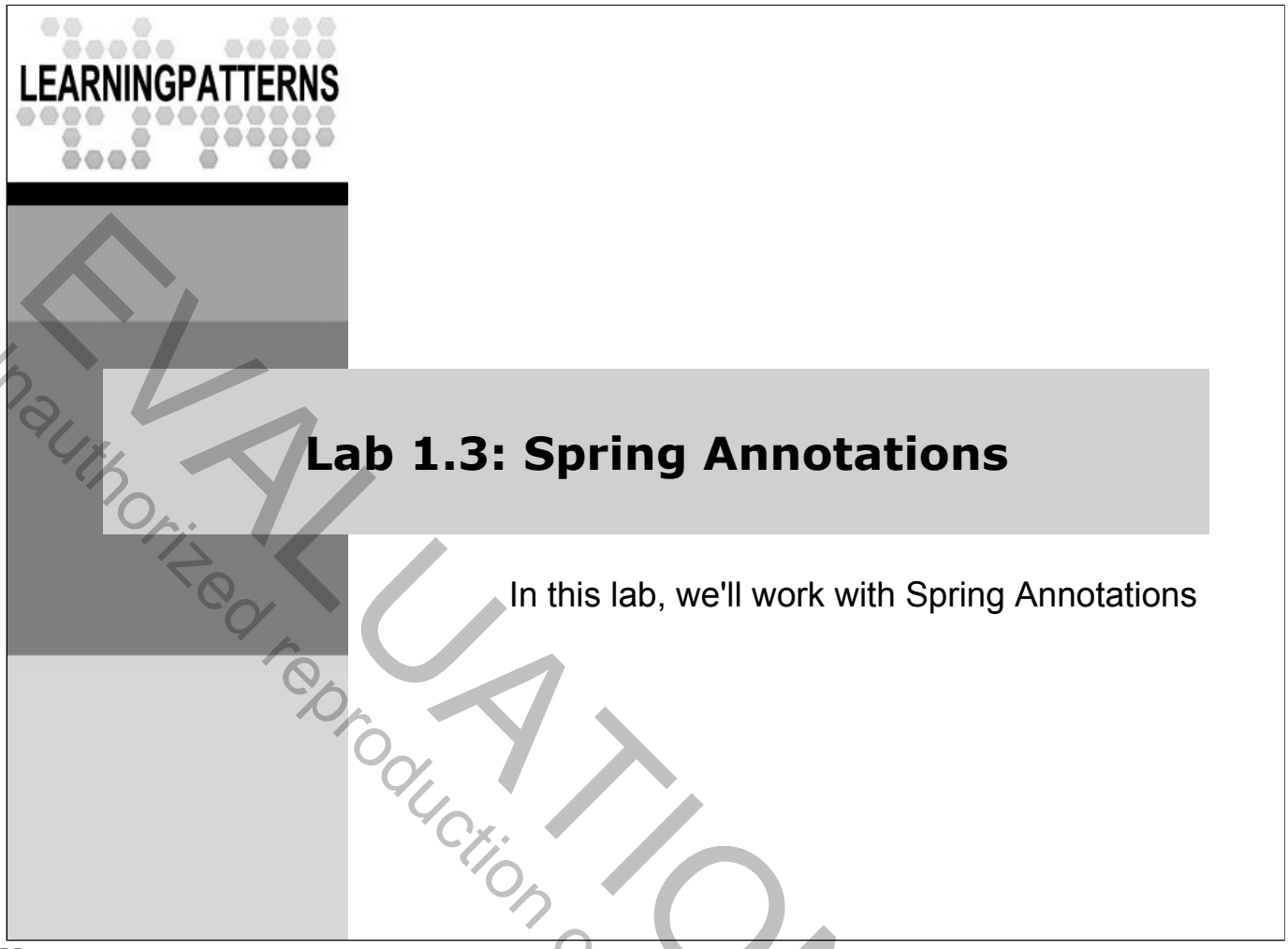
Overview - Spring's XML Schemas

- Spring provides XML Schemas for configuration
  - With custom namespaces and tags with complex behavior (1)
  - e.g. the **`context:`** namespace we just used
- Spring namespaces include:
  - **`aop`**: Configures AOP support
  - **`beans`**: The standard bean tags we've seen already
  - **`context`**: Configures ApplicationContext related things
  - **`jee`**: JEE related configuration such as looking up a JNDI object
  - **`jms`**: Configures JMS related beans
  - **`lang`**: Exposing objects written in language like JRuby as beans
  - **`tool`**: Adds tooling-specific metadata
  - **`tx`**: Configures transaction support
  - **`util`**: Common, utility configuration issues

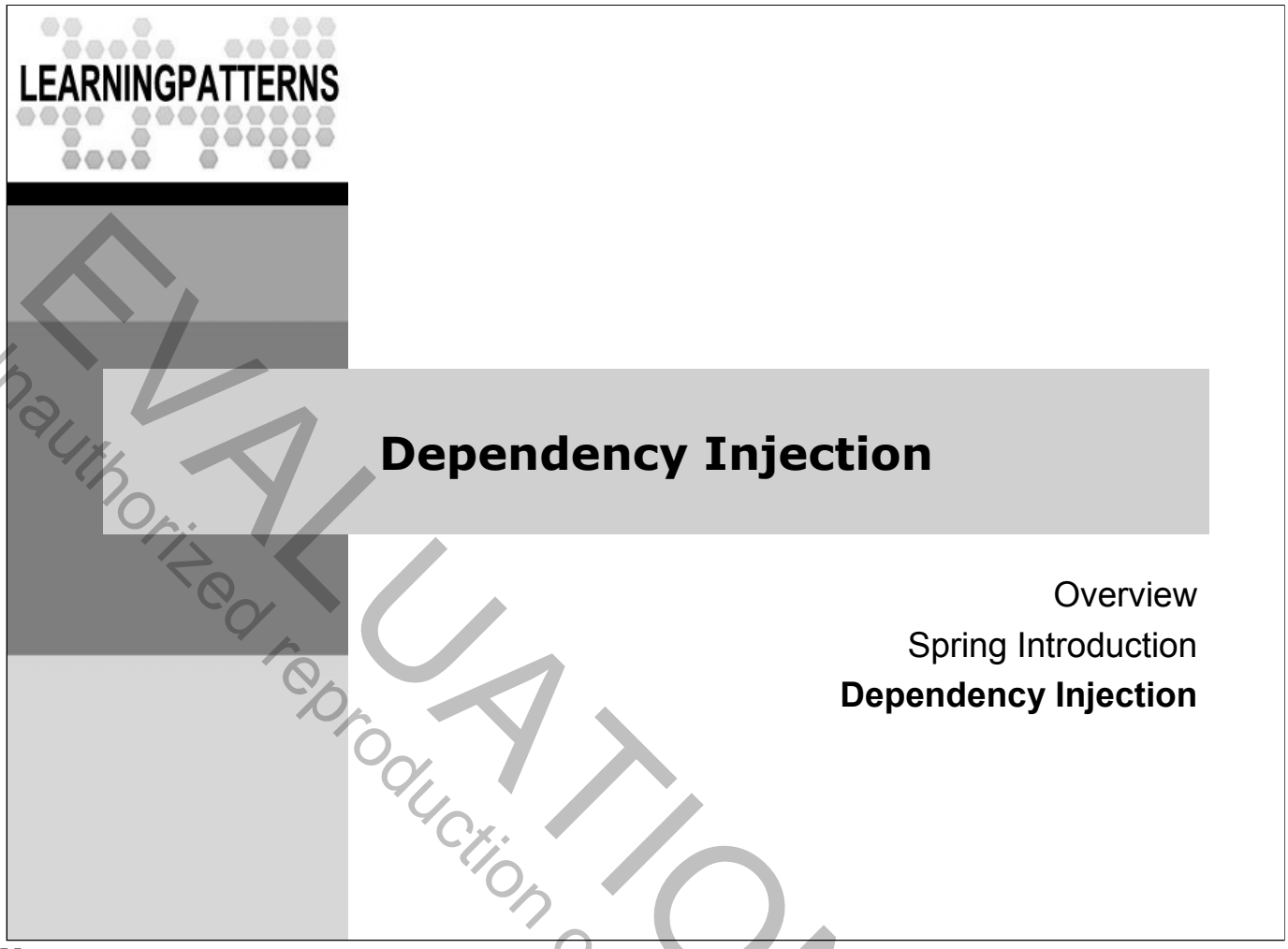
Notes:

(1) We've seen the usage of `<context:component-scan>` previously.

- It is is a shorthand to introduce a number of bean definitions into the configuration.
- The tags in these schemas typically have capabilities beyond simple bean definitions and injection.
- This is one of the nice things about the custom namespaces - they can make configuration much easier.



Notes:



Notes:

Dependencies Between Objects

- In OO systems, multiple objects work together
  - e.g., Object A directly uses Object B to accomplish a goal (1)
  - So Object A **depends on** Object B
- Direct dependencies have several issues
  - **Rigid**: Changes affect other areas, so are hard to make
  - **Fragile**: Changes cause unexpected failures in other areas
  - **Immobile**: Hard to reuse functionality
    - Modules can't be disentangled
- We'll illustrate a direct dependency
  - Then show an alternative approach

Notes:

(1) In straightforward applications, Object A will often just create an instance of Object B and use it.
- Object A is highly coupled to, and directly dependent on, Object B.
- This is initially an easy way to structure a system, but often not the best way.

Example of a Direct Dependency

- CatalogImpl uses InMemoryRepository
  - And creates an InMemoryRepository instance directly
  - CatalogImpl **depends** on the lower level module details
  - To use a different data store - e.g. a FileRepository, CatalogImpl must change (see notes)

```
public class InMemoryItemRepository {
   public MusicItem get(Long id) { /* Details not shown */  }
}
```

```
public class CatalogImpl implements Catalog {
   InMemoryItemRepository rep = new InMemoryItemRepository();
   public MusicItem findById(long ID) {
      return rep.get(id);
   }
}
```

Notes:

- Assume we want to get our information from a class called FileRepository.
  - Assume it also has a get(Long id) method.
- In that case, CatalogImpl might need to be changed to something like that below.

```
public class CatalogImpl implements Catalog {
  FileRepository rep = new FileRepository();
  public MusicItem findById(long id) {
          return rep.get(id);
  }
}
```

  - This is not such a big deal if you have to change it in one place.
  - But imagine if you had to change it in 100 places, or 10,000 places.

- Assume that classes in the examples in this session are in the com.javatunes.service package.
  - We'll be leaving out most package statements in the Java code examples for brevity.

Dependency Inversion

- All modules **depend on abstractions**, not each other
  - In other words - **program to interfaces**
  - `CatalogImpl` only knows about the abstract `ItemRepository`
    - Can be initialized with another type (e.g. `FileItemRepository`)
    - `CatalogImpl` **need not change** - the modules are **decoupled**

```
public interface ItemRepository {
  public MusicItem get(Long id);
}
```

```
// Much detail omitted …
public class InMemoryItemRepository implements ItemRepository { /*…*/ }
```

```
public class CatalogImpl implements Catalog {
   private ItemRepository itemRepository; // get/set methods not shown
   public MusicItem findById(Long id) { return itemRepository.get(id); }
}
```

```
  // Code fragment - most detail omitted
  InMemoryItemRepository rep=new InMemoryItemRepository();
  CatalogImpl catalogImpl=new CatalogImpl();
  catalogImpl.setItemRepository(rep);
  MusicItem found = catalogImpl.findById(1);
```

Notes:

- Dependency Inversion is not a new idea.
  - The idea of "**Programming to Interfaces**" has been around since long before Java.
  - It has been used in non-OO languages also, for example the stdio module in the C programming language abstracted away the details of the actual devices doing the output.
- We'll soon look at Spring's Dependency Injection which makes this design strategy even easier to use.
- We talk of modules here, which in this example result in dependencies between different classes.
- In the example, we see that `CatalogImpl` knows nothing about `InMemoryItemRepository`.
  - It only depends on the `ItemRepository` interface (the abstraction).
  - When `CatalogImpl` is created, you initialize it with an instance of an `ItemRepository` implementation.
  - But it doesn't know any details of this implementation, and doesn't even know its exact type.
- `InMemoryItemRepository` also depends on the abstraction (it implements `ItemRepository`).
  - The abstraction (the interface) is the common language that lets the different parts of the system work together without depending directly on each other.
- We are still creating the `InMemoryItemRepository` directly in our code (in this example).
  - We'll have Spring do this soon, which gives even more benefit.

Dependency Injection (DI) in Spring

- The Spring container **injects** dependencies into a bean
  - Into bean properties, constructors, or via factory method args
- Dependencies **are defined in the Spring configuration**
  - Spring initializes the dependencies ("injects" them) based on the config
  - **No need to explicitly initialize dependencies** in your code
- We illustrate XML config of this below (injecting via a **set method**)
  - **<property …> injects** into the catalog's `itemRepository` property
  - Automatically done when the container creates the bean

```
<beans … > <!-- Much detail / Namespace declarations omitted -->
   <bean id="inMemoryRepository"
      class="com.javatunes.persistence.InMemoryItemRepository"/>
   <bean id="musicCatalog" class="com.javatunes.service.CatalogImpl">
     <property name="itemRepository" ref="inMemoryRepository"/>
   </bean>
</beans>
```

Notes:

Injection with Autowired

- Can also inject with **@Autowired** - shown below for the repository
  - This injects a dependency **by type** - same result as previous XML
- At bottom, we get a catalog from Spring
  - With an already initialized repository (by Spring's DI)
  - We're ready to work !

```
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;

@Component("musicCatalog") // Declares bean - most detail omitted
public class CatalogImpl implements Catalog {
  @Autowired  // can also apply to setter method or constructor
  private ItemRepository itemRepository;
}
```

```
public class CatalogTests {  // imports, other detail omitted
  @Test testCatalogFindById() {
    ClassPathXmlApplicationContext ctx =
      new ClassPathXmlApplicationContext("applicationContext.xml");
    Catalog cat = ctx.getBean(Catalog.class);  // See note (1)
    assertNotNull("item shouldn't be null", cat.findById(1L)); // Use cat
}
```

Notes:

(1) The bean lookup with the code shown directly below works because there is only one bean that implements Catalog, so the container can find this bean by type.

```
ctx.getBean( Catalog.class );
```

–We'll see ways to refine this type of lookup later.

- Note that `CatalogImpl` and `InMemoryItemRepository` didn't need anything special to support Spring's DI.

  –They just need to be written according to the design principles of Dependency Inversion (i.e. coding to an interface, not a concrete type) which is good practice anyway.

  –Once this was done, we didn't need any special capabilities to support DI.

- Note also that there is nothing in your `CatalogTests` code which shows that `CatalogImpl` depends on `InMemoryRepository`.

  –This is defined in your configuration, and handled for you by the container.

@Named/@Inject (JSR-330) Example

- Below, **@Named** is used to declare a bean
  - **@Inject** is used to inject the dependency
- Results are the same as with @Component style
  - And the test client would look exactly the same
- We'll use **@Component** style going forward, as mentioned earlier
  - It's the more common choice

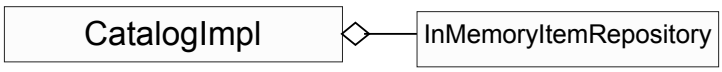
```
import javax.inject.Inject;
import javax.inject.Named;

@Named("musicCatalog") // Declares bean - most detail omitted
public class CatalogImpl implements Catalog {
  @Inject
  private ItemRepository itemRepository;
}
```

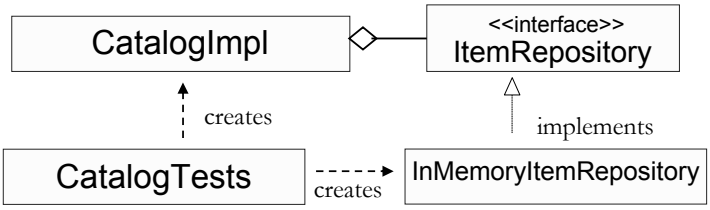
Notes:

Dependency Injection Reduces Coupling

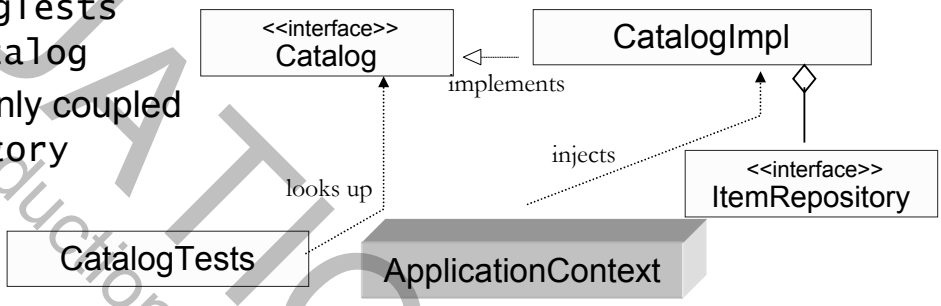
- The simplest case, `CatalogImpl` coupled to `InMemoryItemRepository`



- `CatalogImpl` coupled to `ItemRepository` only (Dependency Inversion)
  - `CatalogTests` coupled to `CatalogImpl` and `InMemoryItemRepository`



- Using DI – `CatalogTests` only coupled to `Catalog`
  - `CatalogImpl` only coupled to `ItemRepository`



Notes:

- In the slide, we show diagrams of the three different ways we structured our code in the earlier slides.
  - In the first, `CatalogImpl` is directly coupled to `InMemoryItemRepository`.
  - In the second, `CatalogTests` is doing the dependency injection of the `InMemoryItemRepository` into `CatalogImpl`.
    - `CatalogTests` is coupled to both of these types, but `CatalogImpl` is only coupled to `ItemRepository`.
  - In the third, Spring is doing the DI, and so `CatalogTests` and `CatalogImpl` are only coupled to abstract interfaces.

- In the Spring DI version, all the dependencies in the code are ONLY to interfaces.
  - This is much better than being coupled to actual implementation classes.
  - It increases flexibility, testability, and ease of maintenance.

Advantages of Dependency Injection

- **Hides dependencies** and **reduces coupling in your code**
  - Coupling is basically a measure of the dependencies
- We see this in two ways:
  - `CatalogImpl` is not coupled to `InMemoryItemRepository`
  - `CatalogTests` is not coupled to `CatalogImpl` or `InMemoryItemRepository`
- The dependencies are still there but **not in the code**
  - Dependencies are **moved to the spring configuration**
  - They're injected into beans without you coding it
  - Commonly referred to as **wiring** beans together
- This leads to **more flexible** code that is **easier to maintain**
  - At a cost – using Spring, and maintaining the spring configuration

Notes:

- Coupling is the measure of how much a module of code relies on other modules.
  - Loosely coupled code is generally considered better code.
- Consider some of the following scenarios, and how DI makes them easier.
  - Testing your code with a testing framework such as JUnit can be much easier with DI – you can configure the application to use mock objects wherever you want – without changing your code at all.
  - Testing different versions of classes can be done just by changing the configuration metadata.
  - In fact, any implementation class that implements the particular interfaces being used can be swapped into your program simply by changing the configuration information.
- There is debate about how worthwhile DI and frameworks like Spring are.
  - However, the principles that it is based on that lead to loose coupling are widely accepted.
  - Spring simply makes it easier to apply these principles.
  - The effort required to learn, adopt, and use Spring is not trivial, but the initial cost of adopting Spring can be well worth it in writing and maintaining any reasonably sized system.

Constructor Injection

- Can easily inject into a constructor (ctor)
  - Assume the `CatalogImpl` constructor shown below
- With XML, **`<constructor-arg>`** means "inject into constructor"
- With **`@Autowired`**, apply it to the constructor
- See notes for some additional detail/capabilities

```
public class CatalogImpl implements Catalog { // Most detail omitted
  public CatalogImpl(ItemRepository repository) { /* … */ }
}
```

```
  <!-- XML config - Other detail omitted -->
  <bean id="musicCatalog"
        class="com.javatunes.service.CatalogImpl">
    <constructor-arg ref="itemRepository"/> <!-- inject via ctor -->
  </bean>
```

```
public class CatalogImpl implements Catalog { // Most detail omitted
  @Autowired // Inject into ctor below
   public CatalogImpl(ItemRepository repository) { /* … */
}
```

Notes:

- For `@Autowired`, if there is only one constructor, you can even leave out `@Autowired`.
  - It will be inferred - but we believe it's better practice to include it explicitly.
- If a constructor has multiple arguments, they will all be injected with `@Autowired`.
  - For example, assuming that a `CatalogImpl` needs an `ItemChecker` to do some sanity checking on items, you might have the following ctor, which will have both arguments injected.

```
@Autowired
public CatalogImpl(ItemRepository rep, ItemChecker checker)
```

- In XML, multiple arguments would be handled with multiple `<constructor-arg>`s.

```
<bean id="musicCatalog" <!-- inject two ctor arguments -->
      class="com.javatunes.service.CatalogImpl">
  <constructor-arg ref="itemRepository"/>
  <constructor-arg ref="itemCheck"/> <!-- itemCheck def not shown -->
</bean>
```

- There are a lot of ways to configure the ctor injection.
  - For example, you can specify the type and/or index of args with XML configuration.
  - These details are best explored as needed in the documentation, not during class time.

Setter Injection vs. Constructor Injection

- **Setter Injection Pros**
  - **Easy**, **Flexible**: Simple setters, easily choose properties to configure
  - Good for **optional properties** (with defaults to reduce not-null checks)
  - Supports **reconfiguration** after startup
- **Setter Injection Cons**
  - **Doesn't guarantee property initialization** (Forget to do it - BUG!)
  - **Setter methods are required**
- **Constructor Injection Pros**
  - **Guaranteed Initialization**: Immediate error if you leave it out (1)
  - Good for **required properties** (can be **immutable** if setter omitted)
- **Constructor Injection Cons**
  - **Unwieldy and Brittle**: Multiple constructors with multiple args unwieldy
    - Refactor if you have this
  - **Less flexible**: Need ctors for each scenario, and can't reset properties

Notes:

(1) If you leave out a constructor argument while configuring a bean, you'll have an error.

- If using XML configuration or `@Autowired`, you'll get a runtime exception as soon as the code is run.

- You can mix constructor and setter injection.
  - For example, using constructor injection on required properties, and setter injection on other properties.

Qualifying Injection by Name

- Consider two repository implementations (first two examples below)
  - What happens if we try to auto-wire one in?
  - The first autowire below **fails at runtime** - which one to inject?
  - The second try is "**qualified**" by the bean name using **`@Qualifier`**
  - It injects based on the bean name

```
@Component // Declare as bean - default id inMemoryItemRepository
public class InMemoryItemRepository implements ItemRepository {…}
```

```
@Component // Declare as bean - default id cloudItemRepository
public class CloudItemRepository implements ItemRepository {…}
```

```
@Component public class CatalogImpl implements Catalog { // Autowire
  @Autowired private ItemRepository itemRepository; // FAILS!
}
```

```
import org.springframework.beans.factory.annotation.Qualifier;

@Component public class CatalogImpl implements Catalog { // Autowire
  @Autowired @Qualifier("cloudItemRepository")
  private ItemRepository itemRepository; // Injects the Cloud-based
}
```

Notes:

- In the first example of injection, where the autowiring is done by type, it fails at runtime.

  `@Autowired private ItemRepository itemRepository; // FAILS!`

  –The container doesn't know which of the two possible types to use.
- In the second example of injection we use `@Qualifier` to supply a bean name.

  –This pinpoints exactly which bean should be injected, and works as you'd expect.

```
@Autowired
@Qualifier("cloudItemRepository")
private ItemRepository itemRepository; // Injects the Cloud-based
```

- There are other, more sophisticated, ways to qualify an injection.

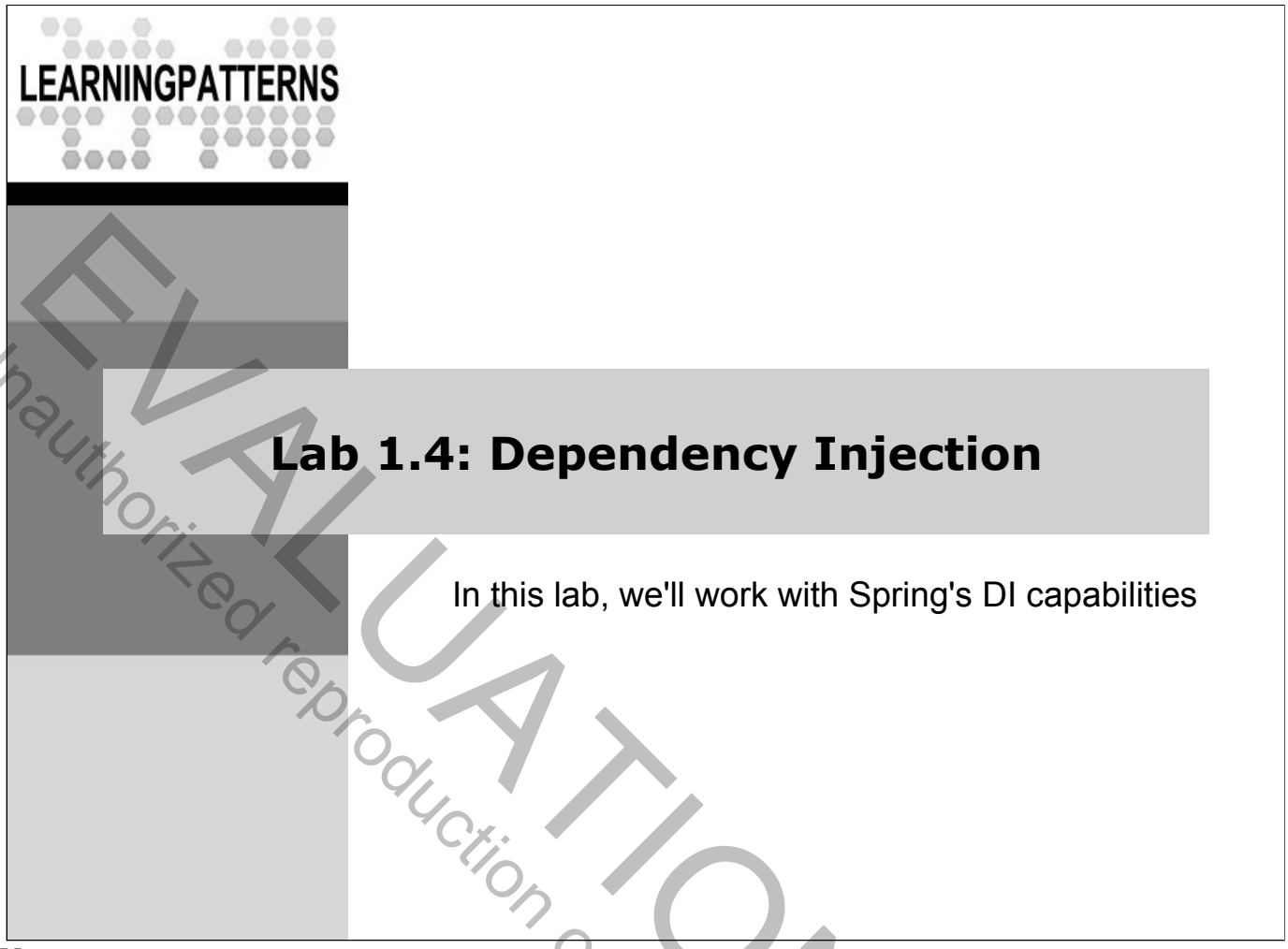
  –This involves defining your own qualifiers, for example `@CloudBased` instead of using names.

  –This is compile-time safe, as it is not a string-based approach, so useful for program correctness.

  –However, it's fairly complex, and arguably the complexity is often not worth the benefit.

  –With adequate testing, name-based qualifying should work fine.

  –In those instances (rare) when it's not suitable, then you can use the more sophisticated approach, which is beyond the scope of this course.



Notes:

Review Questions

- What is Spring, and how does it help you build enterprise apps?
- What is **Dependency Inversion**?
- What is **Dependency Injection**, and how does it work in Spring
- What is an `ApplicationContext`?
- How does Spring use **annotations** on your bean classes for defining and wiring beans?
  - What are the pros/cons of using them?

Notes:

Lesson Summary

- **Spring**: Lightweight enterprise framework that supports:
  - Dependency Injection
  - Persistence support (Repository/DAO and ORM)
  - Integration with standard Web technologies, and MVC Web apps
- **Manages complex dependencies** - non-intrusive and lightweight
  - Supports **loose coupling** between components
  - Encourages **coding to interfaces** (good design)
    - Also called **Dependency Inversion**
- **Uses configuration metadata** to initialize beans and dependencies
  - Via XML configuration files or annotations
- **Dependency Injection**: Injects dependencies based on config
  - Complete **decoupling** from concrete implementation types

Notes:

Lesson Summary

- **`ApplicationContext`**: API to the Spring container functionality
  - Configure/wire beans, access program resources, work with resource bundles, load multiple contexts, and publish events to beans
  - Common implementations include:

    `ClassPathXmlApplicationContext`,

    `FileSystemXmlApplicationContext`, and

    `AnnotationConfigApplicationContext`

- Provides resource access in a flexible and powerful way
  - From file system, the classpath, URL access, and more
  - Supports ant-style wildcards like ***conf/**/ctx.xml***
  - Uses its own **resource classes** to accomplish resource access

Notes:

Introduction to Spring 5 and JPA 2

Lab Manual - Eclipse/Hibernate

LearningPatterns

For more information about Java Enterprise Java, or related courseware, please contact us. Our courses are available globally for license, customization and/or purchase.

LearningPatterns. Inc. Services@learningpatterns.com | www.learningpatterns.com

Global Courseware Services 982 Main St. Ste. 4-167 | Fishkill NY, 12524 USA
212.487.9064 voice and fax

Notes:

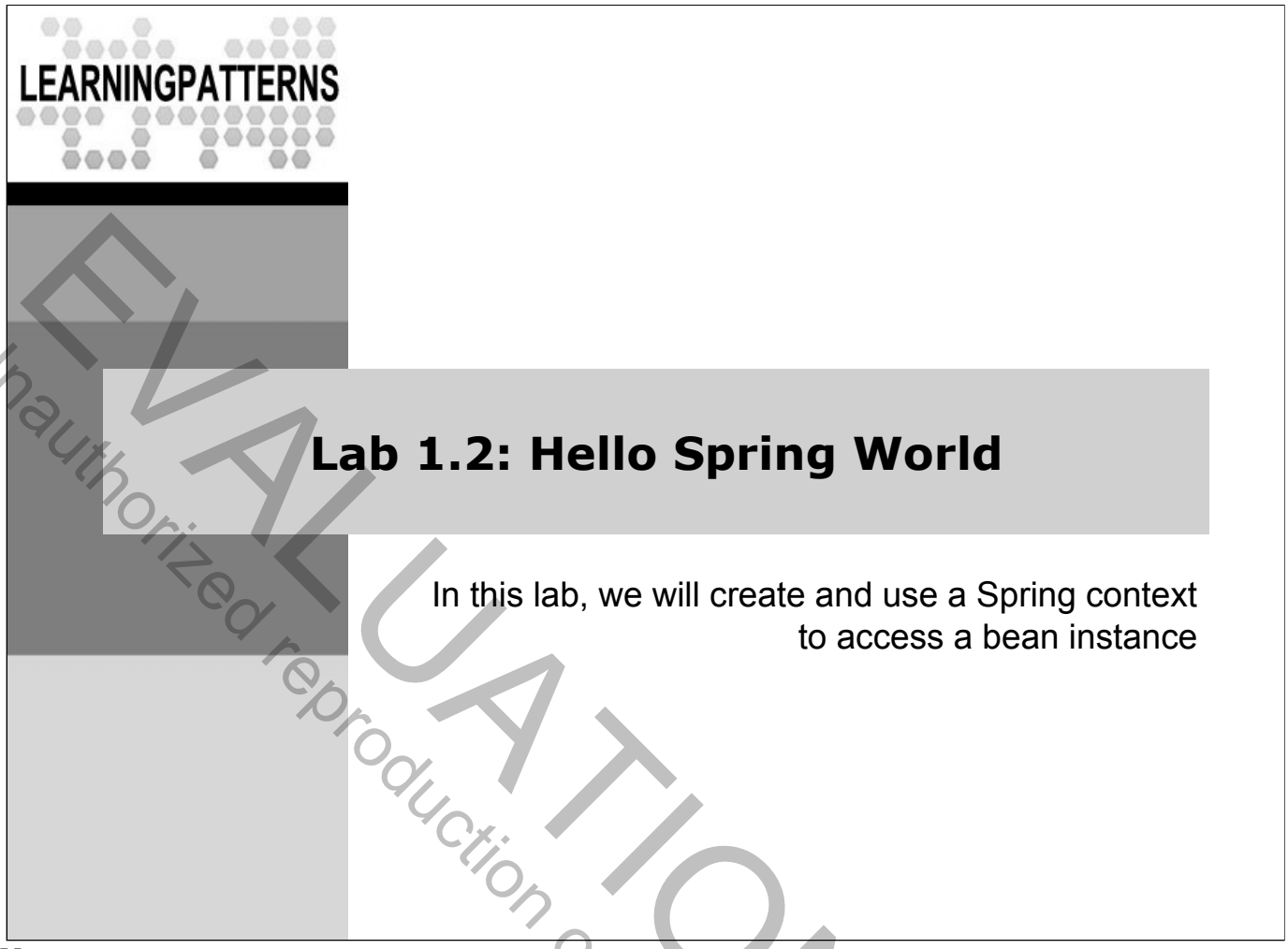
- Version: 20180824
  - Spring Base: 20180521-b
  - JPA Base: 20180820

Release Level

- This manual has been tested with, and contains instructions for, running the labs using the following platforms:
  - **Spring Boot** (tested with 2.0.0.RELEASE)
  - **Spring 5** (tested with 5.0.4.RELEASE)
    - Release used by Spring Boot
  - **JPA 2.2** (Using Hibernate 5.3)
    - We override the Hibernate version used by Spring Boot
  - **Java** (tested with and requires Java 8)
  - **Eclipse Java EE Edition** (tested with Oxygen 4.7.3)
  - **Tomcat** for the Spring/Web material (tested with Tomcat 8.5)
- Recent similar versions of the software will likely work except for potentially small configuration changes

Notes:

- Any 5.x version of Spring should work.
  - The course materials are compatible with all 5.x versions.
  - However, we include some Spring jars and their dependencies with the labs.
  - If you want to use a different version, then it's up to you to make sure you have all the correct Spring jars and the correct dependency jars.
  - We will also introduce Maven and Spring Boot to manage all dependencies later in the course.
- Spring Boot 2 is needed to support Spring 5.
  - At the time of this writing the latest release version was Spring Boot 2.0.0.RELEASE.
- All labs have been tested on Microsoft Windows using the software listed above.
  - Many have been tested on Mac OS also.
  - The labs should work on unix variants with little modification, except for the database setup scripts, which need to use unix shell scripts, which we also supply.



Notes:

Lab Synopsis

- **Overview**: In this lab, we will:
  - Become familiar with the different parts of basic Spring
  - Create and use a Spring context to access a bean instance
  - Write and run a simple Spring test
- **Builds on previous labs**: None
- **Approximate Time**: 20-30 minutes

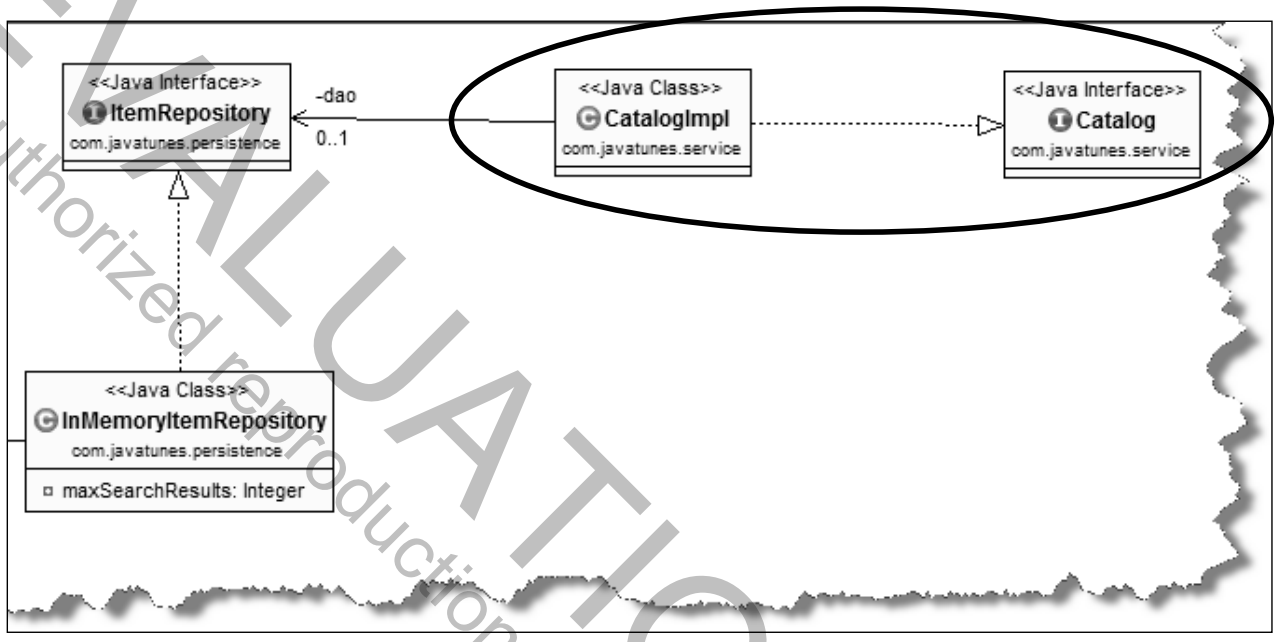
Notes:

- The purpose of this lab is to become familiar with the different parts of a Spring application.
  - Accordingly, the program is as simple as we can make it.
  - It mirrors the code already shown in the manual pages.
- In later labs, we'll work with more complex applications.

Object Model

Lab

- **Object Model**: Our focus will be on the following types
  - We'll cover more types shortly
  - Note: Search methods in the catalog won't work yet



Notes:

Lab Preparation

- The new lab folder where you will do all your work is:
  workspace\Lab01.2

Tasks to Perform

- **Close** all open files and projects
- **Create a new Java project** called Lab01.2 in the workspace
  - See Lab 1.1 instructions on how to do this if you need to
  - Remember to **add the Spring user library**
  - You can expect to see **compiler errors** until the lab is completed.

Notes:

- If you need to add the Spring Library after creating the project, do the following.
- **Right Click** on the project in Package Explorer, select **Build Path | Add Libraries.**
  - You should be in the **Libraries** tab, so click the **Add Library** button, and in the dialog , select **User Library** then click **Next.**
  - Check off **Spring**, then click **Finish** in all open dialogs.

Writing and Configuring a Bean in Spring

Lab

Tasks to Perform

- **Open `CatalogImpl`** for editing
  - The class is in the `com.javatunes.service` package, under ***src***
  - Make sure **`CatalogImpl`** implements the **`Catalog`** interface
  - Remember - we code to interfaces to decouple from a specific implementation
  - **Save** your changes
- **Open *applicationContext.xml*** in the *src* folder for editing
  - Click the **Source tab** to edit
  - Finish up the declaration of the <bean> element by declaring:
    - An id of **`musicCatalog`**
    - A class of **`com.javatunes.service.CatalogImpl`**
  - **Save** your changes

Notes:

- There are a number of other types in the src tree that we will use in future labs.
  - Ignore these for now.

Using a Bean in a Program

Tasks to Perform

- **Open** `com.javatunes.service.CatalogTest` (*test* folder) and:
  - In `testCatalogLookupPositive()`, modify the constructor for `ClassPathXmlApplicationContext`
    - Specify the `applicationContext.xml` file.
  - Next, look up the `musicCatalog` bean (by type) from the context [1]
    - `Catalog catalog = ctx…`
  - Make sure the catalog bean is not null (use `assertNotNull`)
  - Output the catalog bean (Just use `System.out.println()`)
  - Finally, call `close()` on the context and fix any compilation errors
- **Run** *CatalogTest* as a unit test and make sure the test passes
  - **Right click | Run As …| JUnit Test** as in previous labs
  - Your tests should pass, and you should see some console output
- You've successfully configured and used a bean with Spring
  - Congratulations !

Notes:

[1] Since there is only one bean that implements `Catalog`, we can look it up by type.

- Lookup by type is preferred as long as it's possible.

Logging and Additional Things (Optional)

Lab

Tasks to Perform

- **Open** *log4j2.xml* in the *src folder*
  - This configures some of the logging that Spring will do
  - The root logger is configured at **`error`** level, spring at **`info`**
  - You can decrease Spring logging by configuring it at **`warn`**

```
<Logger name="org.springframework" level="warn"/>
```

  - **debug** level increases the logging - try some levels
- **[Optional]** Other things to try
  - Try looking up the bean by name, – what happens? What about if you look it up by the wrong name?
  - Try looking up a `CatalogImpl` instead of a `Catalog`
    - Does this work? Is it a good idea? (see notes)
  - Change your code back to your original solution before proceeding

STOP

Notes:

- Spring has logging built into it.
  - It uses a custom logging bridge (built on Apache Commons Logging).
  - It will detect the log4j2 jar files and use that if it is on the classpath.
    - It defaults to using JUL (`java.util.logging` - part of the JDK)
  - You can see the logging by properly configuring it, as we do in the *log4j2.xml* file.
  - The details of log4j2 are beyond the scope of this course.

- Looking the bean up as a `CatalogImpl` will work.
  - These are POJOs, and the actual type is `CatalogImpl`.
  - It's not a good idea though, because you are defeating the purpose of decoupling your code from the actual implementation type.

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
Ph: 303-302-5280
www.ITCourseware.com