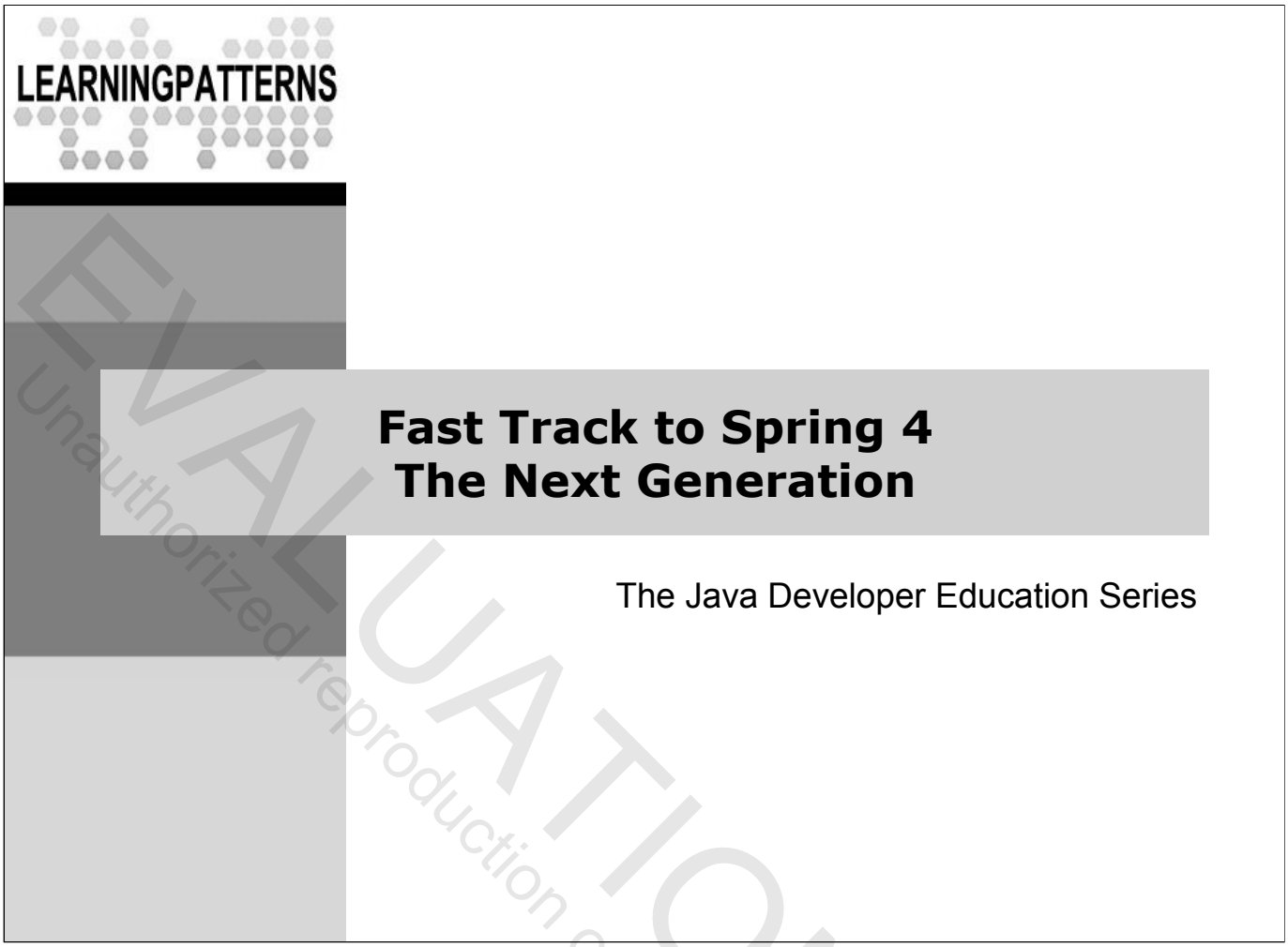
Table of Contents: Fast Track to Spring 4



Notes:

Workshop Overview

- An in-depth course teaching the use of Spring 4 to build enterprise applications using Java
  - We also demonstrate best practices for newer areas of Spring/Java technology
- The course covers the following areas of Spring technology
  - **Core features of Spring**
  - **Data Access Features**
  - **Aspect Oriented Programming (AOP)**
  - **Transaction Support**
  - **Web Application Support**

Notes:

Workshop Objectives

- Understand the Spring framework and use its capabilities, including:
- **Spring Core**: Dependency Injection (DI) and lifecycle management of application objects
  - Spring configuration and API for writing Spring programs
  - XML, Java-based, and annotation-based config
- **Data Access**: Data access via Spring's data support
  - DataSource support
  - Hibernate and JPA-based Repositories/DAOs
- **AOP**: Spring's AOP (Aspect Oriented Programming) capabilities for injecting crosscutting concerns
- **Transactions**: Controlling transactions declaratively with Spring
  - Via both Spring annotations and XML configuration
- **Web**: Integrating Spring with Web applications

Notes:

Workshop Agenda

- Session 1: **Introduction to Spring**
- Session 2: **Configuration in Depth**
- Session 3: **Wiring in Depth**
- Session 4: **Database Access With Spring**
- Session 5: **Aspect Oriented Programming (AOP)**
- Session 6: **Transactions**
- Session 7: **Web Applications with Spring**
- [Optional] Session 8: **XML Specific Configuration**

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:

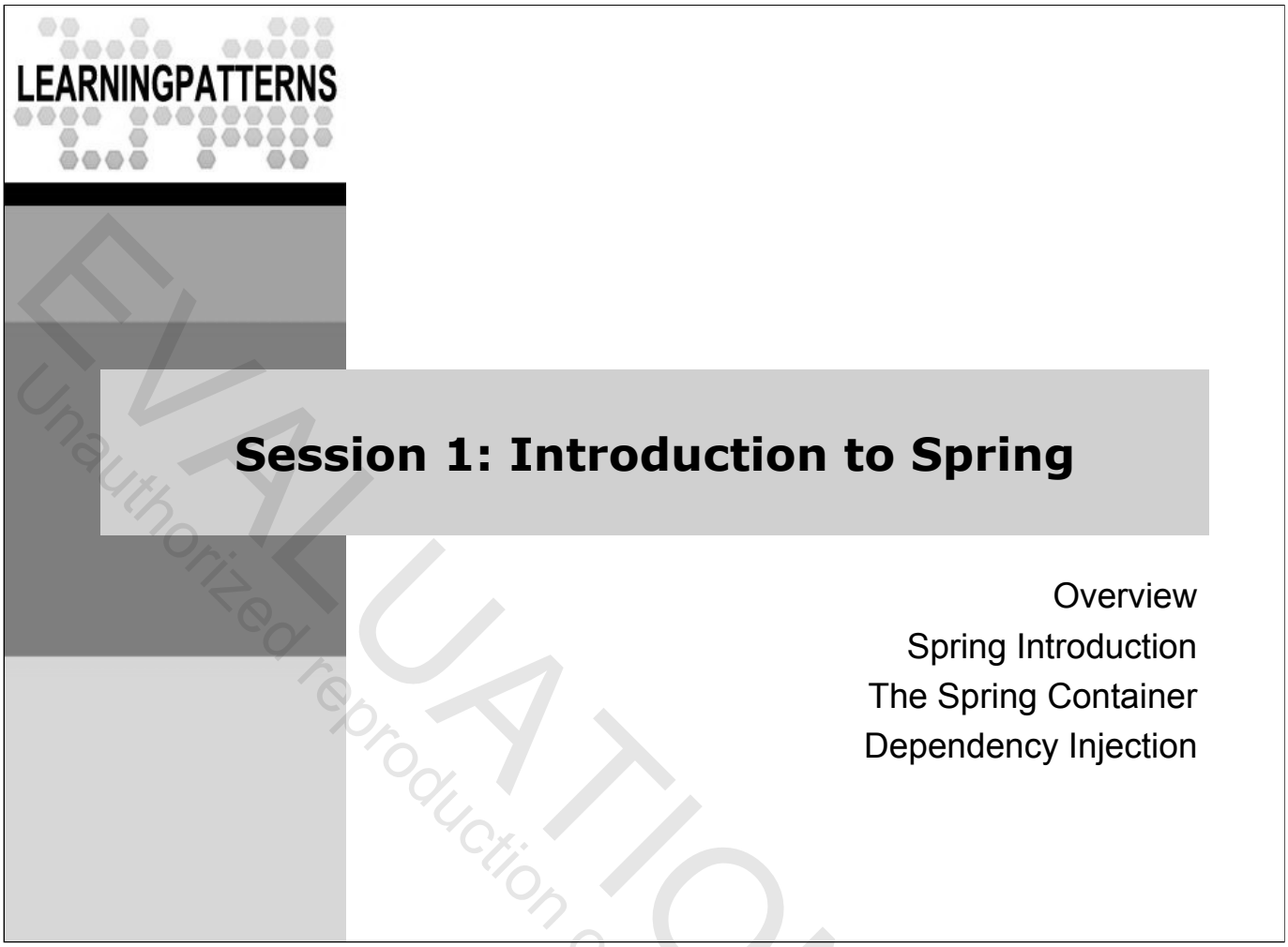
- (1) If we had additional information about a particular item in the slide, it would appear here in the notes

- 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:

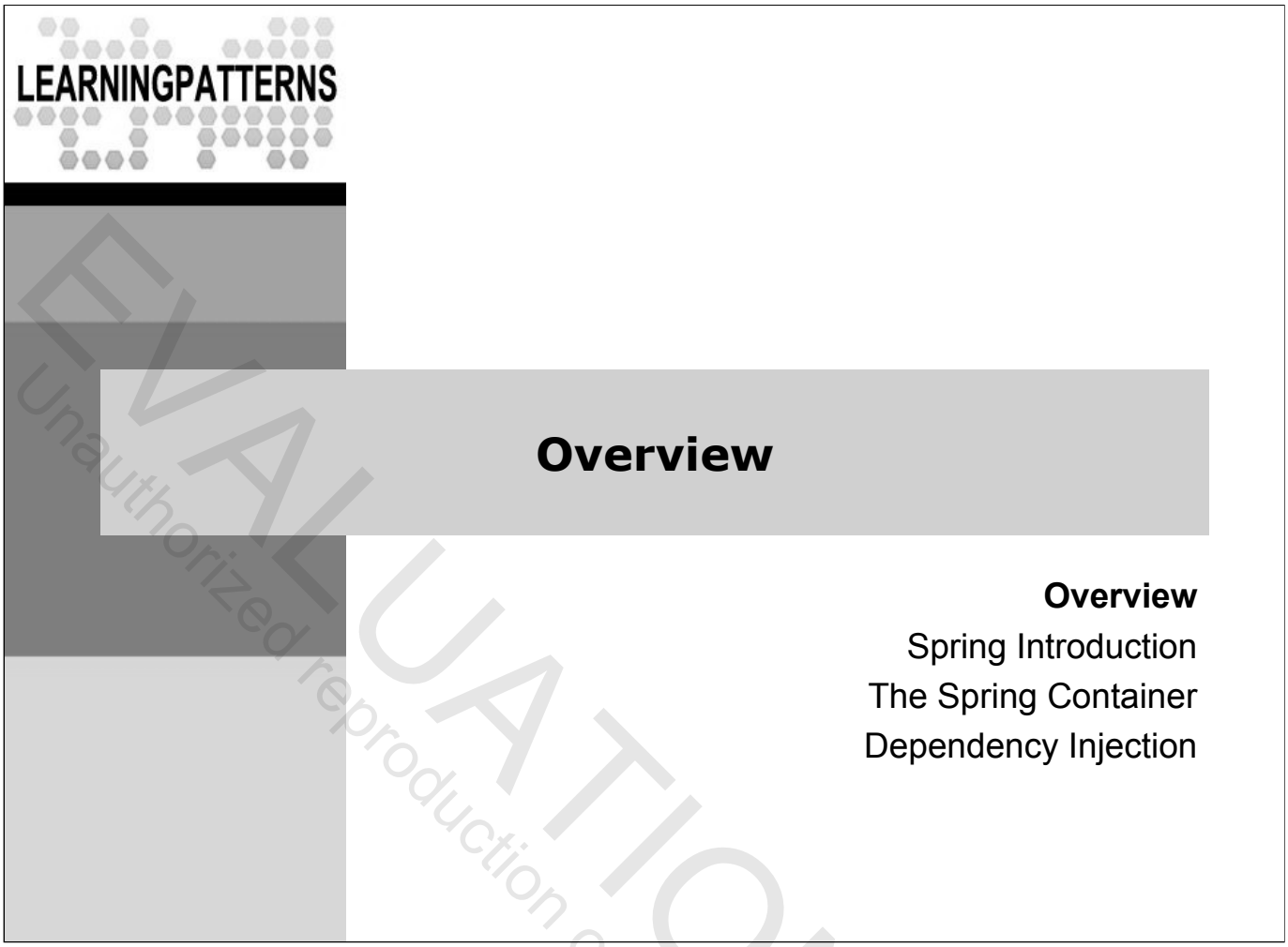


Notes:

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:

The Challenge of Enterprise Applications

- **Enterprise applications** are complex
  - Many **application types**, with complex dependencies
  - **Persistent data** retrieved from / stored to a data store
  - **Transactional** requirements
  - **Remote access** requirements (distributed apps)
  - **Web access** requirements
- Spring supports all these requirements
  - They are the motivation behind Spring
  - We'll start with managing object lifecycle and dependencies
    - A core requirement
  - We'll cover other capabilities later

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

Spring vs. JEE

- **JEE** (Java Enterprise Edition) also supports enterprise apps
  - Current JEE standards match many Spring capabilities
  - e.g. **CDI** (Context and Dependency Injection) provides lifecycle and dependency management
  - This is an evolution in JEE capabilities
- There has been a lot of conversation about Spring vs. JEE (1)
  - Strengths / weaknesses, which to use, etc.
  - Both are powerful frameworks - choose your technology based on your requirements
    - It may be dictated by existing constraints
  - You may use both - fitting the best solution to your needs

Notes:

- (1) Current releases of Java/JEE offer many of the core capabilities of Spring
  - Some people can feel very strongly toward one or the other
  - However, it is not productive to try to compare them in an absolute sense (e.g. Spring beats the heck out of JEE in every way)
  - Generally, there are many constraints and requirements that will guide or dictate your choice of technology to use
  - We will not engage here in trying to declare a "winner" in some technology battle
  - Rather, we'll present the capabilities that Spring has to offer, it's strengths as well as weaknesses, so you can make an informed decision to meet your needs well

What is Spring?

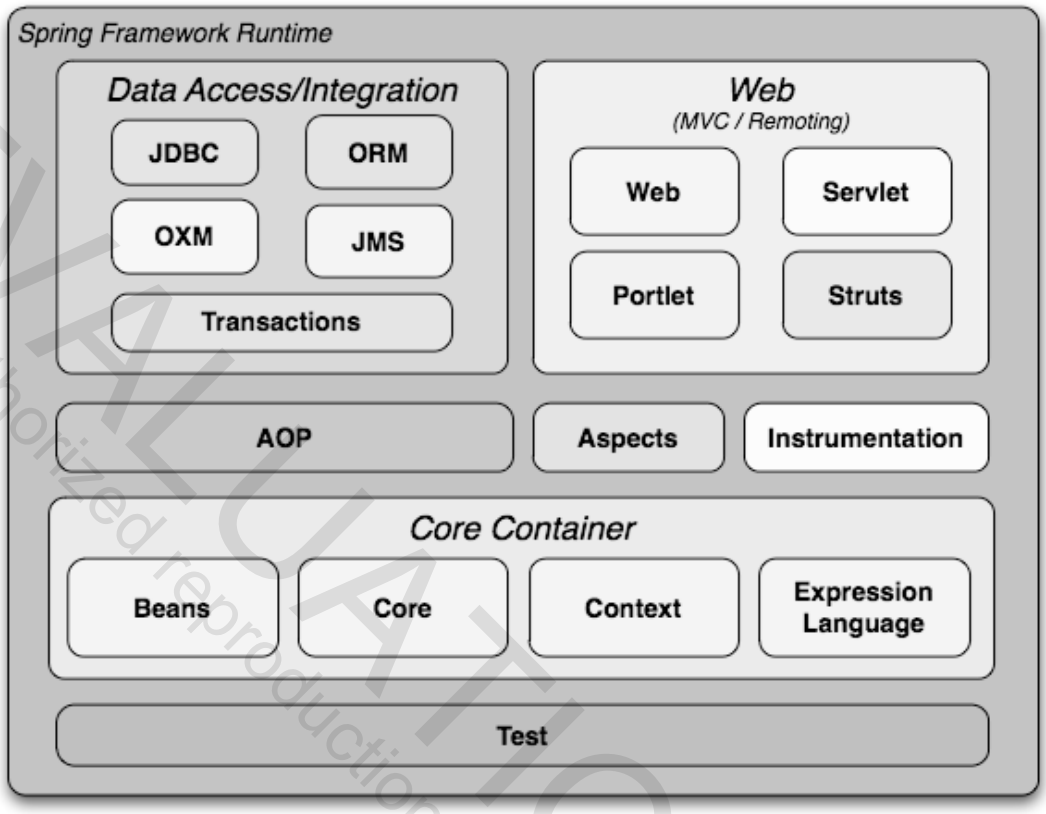
- **Lightweight framework to build enterprise applications**
  - Non-intrusive in your programming
  - Lets you use only what you need
  - But still provides many advanced capabilities
- Capabilities include:
  - **Dependency Injection** (**Inversion of Control/IOC**) container to manage objects and their dependencies
  - **DAO/Repository** package that simplifies database use
  - **ORM** package integrating with persistence technologies (e.g. JPA)
  - **AOP** package for aspect-oriented programming
  - **Web** package integrating with Web technologies
  - **MVC** package providing a Model-View-Controller Web framework
  - A **security** framework for authentication and authorization

Notes:

- DAO stands for Data Access Object
- ORM stands for Object-Relational Mapping

- 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

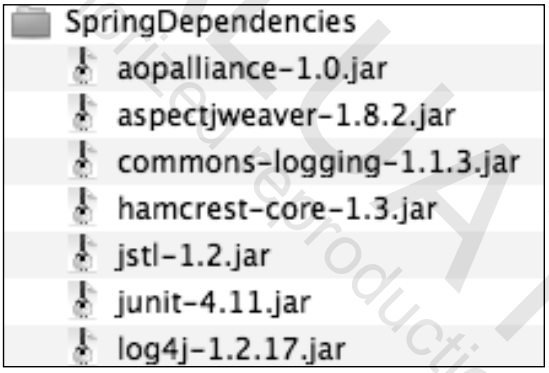
- Spring home page: **http://spring.io/**
- Spring framework packaged as a set of jar files
  - Each module packaged in its own archive
    - e.g. *spring-beans-4.1.0.RELEASE.jar*
- Spring also has dependencies on external technologies (1)
  - e.g. - Apache commons logging, jars for AOP, and more
- Dependencies usually provided by a build management tool
  - e.g. **maven**, which can automatically download dependencies
  - We provide all lab dependencies in the lab setup
  - Maven is covered briefly in an Appendix

Notes:

- In the past, 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 just the Apache commons 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

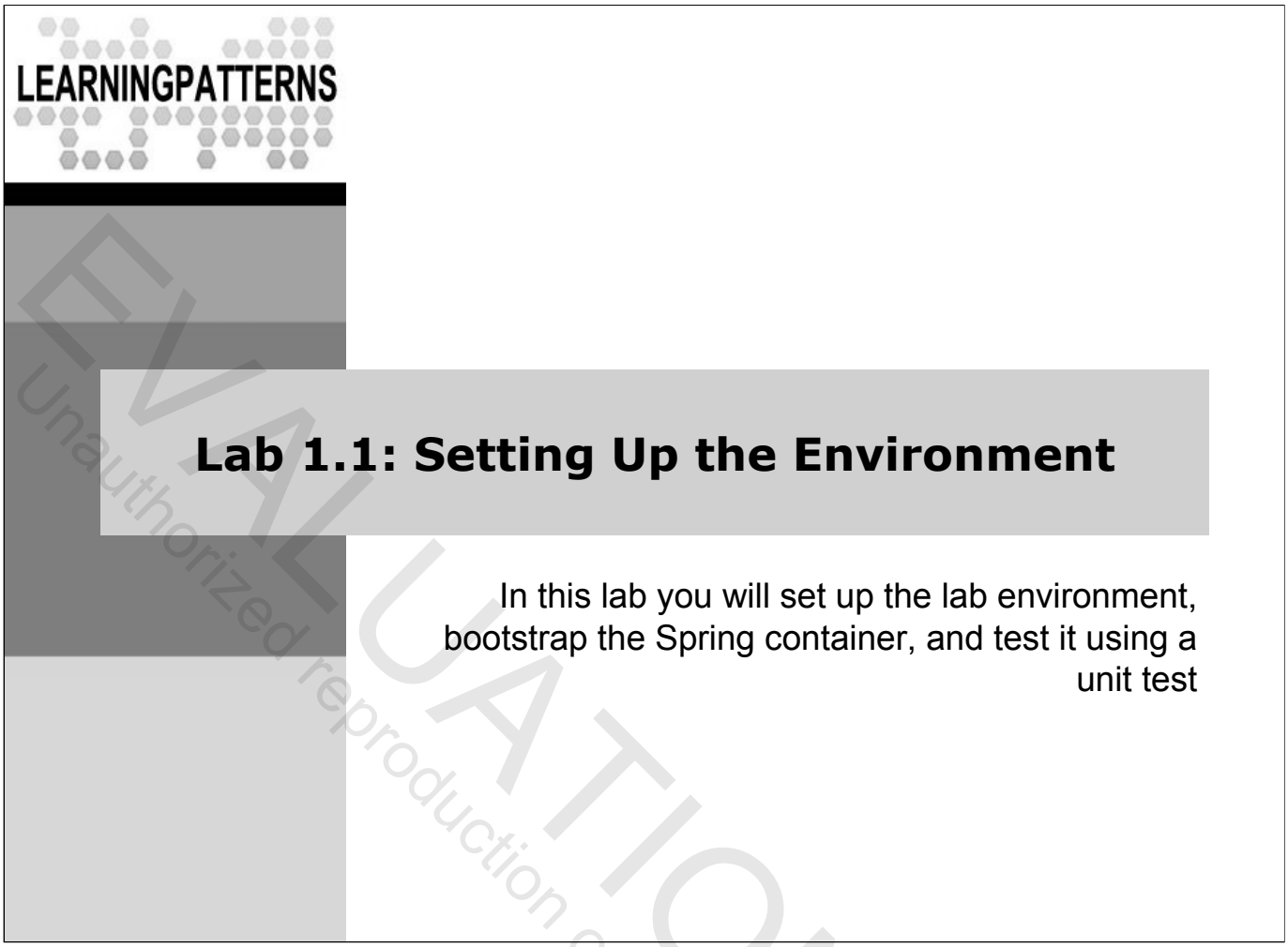
The Spring jars

- At right, are the libraries we supply for the labs (1)
- Below, are the external dependencies (2)
  - We downloaded these all using maven

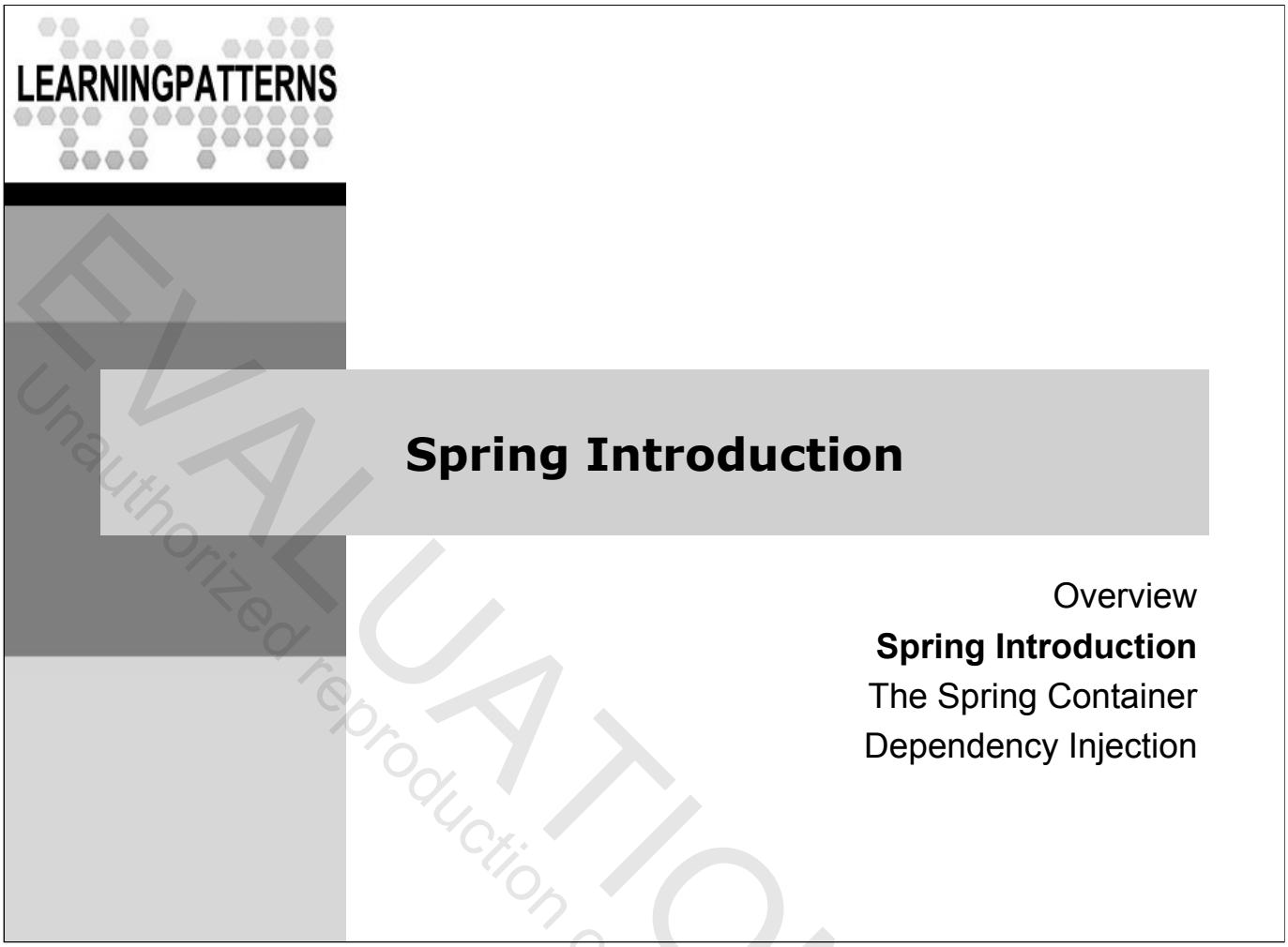


Notes:

- (1) We supply most of the Spring project jars in the lab setup
  - We downloaded these using maven and a *pom.xml* specifying the different modules available in Spring (e.g. spring-aspects, spring-jdbc) 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 needed
  - 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
  - We also supply all the Hibernate/JPA dependencies for the JPA labs in a separate folder
  - These contain many more jars



Notes:



Notes:

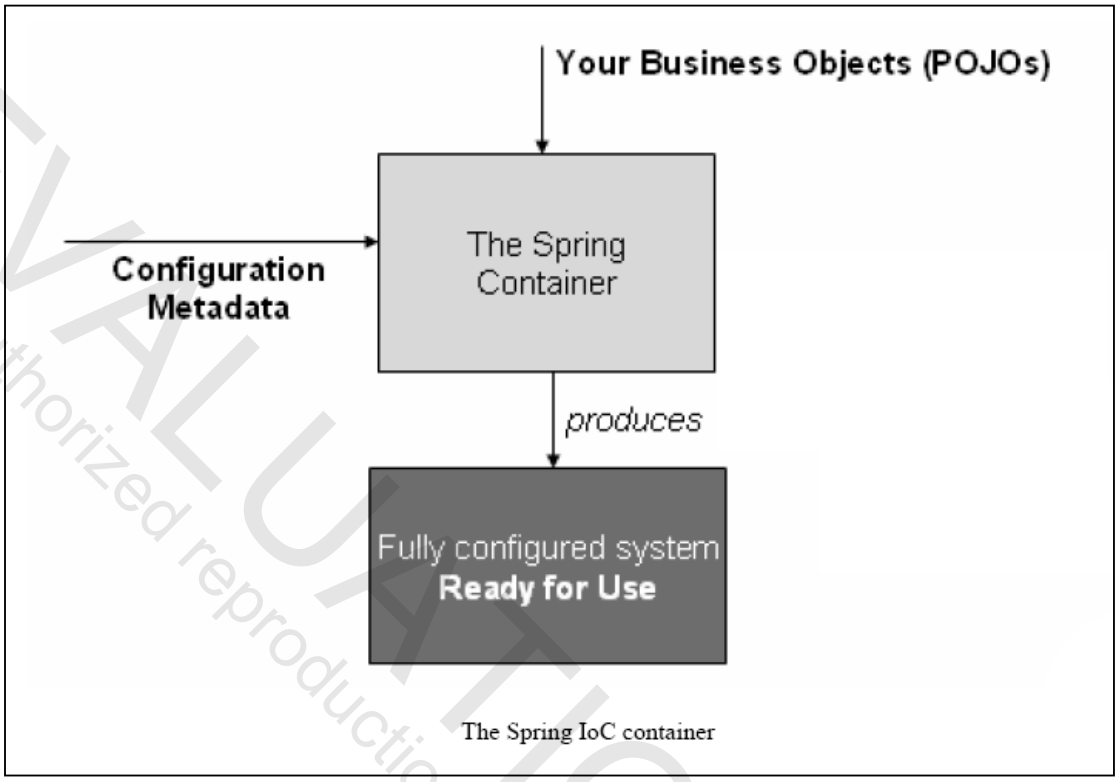
Managing Beans

- **Managing application objects** is a core Spring capability
- Managed objects are called **beans** (1)
  - But they're just **POJO**s (Plain Old Java Objects)
- The **Spring container** is the "manager"
  - Also called the **Dependency Injection (DI)** or **Inversion of Control** (**IoC)** container
  - Uses **configuration information** (metadata) to **instantiate, configure, and assemble** beans
  - Config styles include two kinds of Java annotations and XML
- **Bean definitions / dependencies** done via configuration (2)
  - Container uses this to create and manage beans

Notes:

- (1) The term bean doesn't mean that much - it's a common name (e.g. JavaBeans and EJB)
  - It basically means an object managed by the Spring container in this context
- (2) 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
- We'll cover the Spring container, and the principles behind it, in more detail later
  - For now, we will just describe some of its basic capabilities
- 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 int size();
}
```

```
package com.javatunes.service;

public class CatalogImpl implements Catalog { // Detail omitted

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

Configuration Metadata

- **Metadata**: Information about your beans
  - i.e. Configuration information
  - Early on, only XML metadata was available
  - Annotation-based approaches now available
- We'll start with the XML configuration
  - Still widely used in existing applications, so good to know
  - We'll cover annotations also, as a lot of new work uses them
- Once you understand Spring, all the configuration types are fairly straightforward (1)

Notes:

- (1) Once you know how Spring works, any of the configuration styles become easily understandable
  - The question then becomes - which is more applicable to your system
  - We'll discuss that as we cover each option

XML Bean Definitions

- Bean definitions configure objects managed by Spring
  - Typically **many** bean definitions and dependencies
- A top level **<beans>** element contains bean definitions
  - <beans> contains namespace declarations and **<bean>** elements as shown below
  - The config file is often named ***applicationContext.xml*** (1)
    - But can have any name

```
<?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
- 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 file
  - The latest version will automatically be picked up
  - For example, spring-beans-4.1.xsd
- Typically you'll configure beans such as
  - Service layer objects, Respository objects, Hibernate session factories, JMS queue references
- Some `BeanFactory` implementations also permit the registration of existing objects that have been created outside the factory (by user code)

Declaring Beans

- Bean definitions specify a **package-qualified class name**
  - Generally the bean implementation class - what the container instantiates (1)
  - Below, the **`class`** attribute specifies our implementation - **`com.javatunes.service.CatalogImpl`**
- In general, a **bean identifier** (or name) is specified
  - It's a bean label – available in code and configuration
  - A bean can have multiple names (aliases)
  - Below, the **`id`** specifies **`musicCatalog`**
  - Generally, bean names use `camelCase`

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


Notes:

- (1) 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
- If you don't specify a bean name, the container will generate a unique name for the bean
  - This can be useful in some more sophisticated scenarios
- The `id` attribute must follow the standard XML rules for ids
  - There is a limited set of characters that are allowed for XML ids
- The `name` attribute can also be used to specify (multiple) names
  - You can have a value that specifies multiple bean names separated by a comma, semicolon, or whitespace
  - If there is no `id` attribute, the first name becomes the identifier, the rest are aliases

```
<bean name="musicCatalog, myCatalog"
                        class="com.javatunes.service.CatalogImpl"/>
```

  - You may use the `id` and `name` attributes at the same time
    - This is generally not done - it can be confusing, and brittle to maintain

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 will use shortly
- 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'll soon see how some of the tags in the context namespace work
  - In particular, the `<context:annotation-config/>` 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
- We'll see examples of a number of these later in the course

Notes:

The Spring Container

- The Spring container
  - Provides a **configuration mechanism** for objects (config files)
  - **Instantiates/initializes** application objects
  - **Resolves object dependencies** based on configuration
- `org.springframework.context.`**ApplicationContext**
  - Core API to **access the container** in your code
  - Multiple implementations provided for flexibility, e.g.
  - **`ClassPathXmlApplicationContext`**: Common implementation loading 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`

Working With Spring

- High level scenario for using Spring includes the following:
- **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
- **Retrieve beans** via the context instance and use them
  - e.g. use **`getBean()`** to look up a bean by type or name

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

A Simple Spring Example (1 of 2)

- The code below provides a simple Spring example
  - Instantiates a **`ClassPathXmlApplicationContext`** to read *applicationContext.xml* from the classpath
    - This creates the spring container
    - Note: The no-arg constructor will look for *applicationContext.xml* by default
  - Uses a JUnit assert to verify the container isn't null (JUnit details soon)

```
package com.javatunes.domain;

import static org.junit.Assert.*;
import org.junit.Test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class UT_Catalog {
   @Test
   public void springTest() {
      ClassPathXmlApplicationContext ctx=
            new ClassPathXmlApplicationContext("applicationContext.xml");
      assertTrue("spring container should not be null", ctx != null);

      // Continued on next slide
```

Notes:

- `ClassPathXmlApplicationContext(String)` loads the named file from the classpath
  - If you don't supply a name, "*applicationContext.xml*" is used
  - The context implementation uses Spring-based resource classes (`org.springframework.core.io.Resource` implementations) to access the file
  - The resource classes abstract access to resources - often used by other classes to access data
  - They allow you to flexibly specify paths to resources as resource strings
  - These strings are then mapped by a particular implementation to the actual resource, for example a file on a file system or from the Java classpath
  - `ClassPathResource` is a concrete implementation that allows you to easily access resources on the classpath, e.g.

    ```
    Resource resource = new ClassPathResource("applicationContext.xml");
    ```

  - `ClassPathXmlApplicationContext` accesses its resources like this under the hood

- A bean definition can be seen as a recipe for creating one or more actual objects
  - The container looks at the recipe for a particular bean when asked for it, and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object

A Simple Spring Example (2 of 2)

- **Our code**:
  - Prints out the Spring environment
  - Looks up a catalog bean (by type) from the container
    - Assumes only one bean matching the `Catalog` type
    - Preferred over lookup by name - more on this later
  - Uses the catalog to look up an item
  - Closes the context (destroying all beans)
- The example looks like a lot of work to create a single object !
  - Let's examine what benefit it gives

```
      System.out.println("Spring was bootstrapped for environment " +
           ctx.getEnvironment());

      // Note that getBean uses Java generics (no casting needed)
      Catalog cat = ctx.getBean(Catalog.class);
      MusicItem item = catalog.findById(1L); // Use our bean
      ctx.close();
   }
}
```

Notes:

Why Bother?

- Main benefit: We've **decoupled** our code **from a dependency on the implementation** class `CatalogImpl`
  - Our code doesn't know about `CatalogImpl`
  - It just knows about needed functionality (interface `Catalog`)
- Can use any conforming implementation in our configuration
  - Client code **will not change**
  - Which is why we **code to interfaces**, not concrete types
  - Very useful for complex systems
- We'll soon see additional capabilities that make Spring useful

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

ApplicationContext Interface

- The **`org.springframework.context.ApplicationContext`** interface extends `BeanFactory`
- Exposes large number of Spring's 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**
- Generally use `ApplicationContext` (not `BeanFactory`)
  - This is the recommended practice
  - It builds on `BeanFactory`, and adds many additional capabilities

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

Common ApplicationContext Implementations

- **`ClassPathXmlApplicationContext`**: Standalone context loading XML config files from the classpath
- **`FileSystemXmlApplicationContext`**: Standalone context loading XML config files from the file system or URLs
  - Both in package `org.springframework.context.support`
- **`AnnotationConfigApplicationContext`**: Standalone context accepting annotated classes as input
  - In package `org.springframework.context.annotation`

Notes:

- There are also Web-based application contexts we'll cover later
- We'll cover the ApplicationContext API in various parts of the course

Using an ApplicationContext

- Below, **`ClassPathXmlApplicationContext`** loads a config file
  - *applicationContext.xml* **must be in the classpath**
  - It's easy to load multiple files – as shown at bottom
  - There are many constructors supporting different arguments (1)

```
package com.javatunes.teach;
import org.springframework.context.support.ClassPathXmlApplicationContext;
@Test
public class UT_Spring {
   public void smokeTest() {
      ClassPathXmlApplicationContext ctx =
          new ClassPathXmlApplicationContext("applicationContext.xml");
      assertTrue("spring container should not be null", ctx != null);
      ctx.close();
   }
}
```

```
// Load configuration from two files
ApplicationContext ctx =
    new ClassPathXmlApplicationContext( "ctx-1.xml", "ctx-2.xml" );
```

Notes:

- (1) Some constructors include:
  - `(String… configLocation)`: Load configuration from the given XML file(s)
    - Uses Java varargs - pass in any number of comma separated strings or array or string
  - `(String[] configLocations)`: Same as above
- When the configLocation(s) do not have a prefix (e.g. file:), then the actual Resource type used to load the bean definitions depends on the specific application context
  - e.g. – a `ClassPathResource` for a `ClassPathXmlApplicationContext`
- For `FileSystemXmlApplicationContext`:
  - For historical reasons, plain paths are interpreted as relative to the current VM working directory, even if they start with a slash. (Consistent with the semantics in a Servlet container.)
  - This means that the following are equivalent
    - `new FileSystemXmlApplicationContext( "applicationContext.xml" )`
    - `new FileSystemXmlApplicationContext( "/applicationContext.xml" )`
  - Use an explicit "**file:**" prefix to enforce an absolute file path.
    - `new FileSystemXmlApplicationContext( "file:/applicationContext.xml" )`
  - Note that if you use a `FileSystemResource` instance directly, then this behavior is different – it behaves as you would expect, with a leading slash designating an absolute path

Specifying Configuration Files

- ant style wildcards can be used
  - **conf/**/ctx.xml**: All *ctx.xml* files under any subdir of conf
  - **conf/*-context.xml**: All files in conf, ending in –context.xml
- Can also use the **file:** and **classpath:** prefixes
  - Forces use of the specified loading mechanism
  - e.g. The following loads definitions from the classpath
    - Even though `FileSystemXmlApplicationContext` is used

```
new FileSystemXmlApplicationContext("classpath:ctx.xml");
```

- Spring uses its **Resource** classes under the hood to access the config info for `ClassPathXmlApplicationContext`
  - This functionality is part of Spring

Notes:

- In the case of `ClassPathXmlApplicationContext` Spring uses a `ClassPathResource` to access the configuration files
  - Resources are used internally by Spring in many places
- The documentation provides this info about the classpath prefix
  - *This special prefix specifies that all classpath resources that match the given name must be obtained (internally, this essentially happens via a ClassLoader.getResources(...) call), and then merged to form the final application context definition*
  - *The wildcard classpath relies on the getResources() method of the underlying classloader. As most application servers nowadays supply their own classloader implementation, the behavior might differ especially when dealing with jar files.* [Spring Reference Documentation]
- The Spring resource classes have quite a bit more functionality than the plain Java URL class

Some BeanFactory/ApplicationContext API

- **`boolean containsBean(String)`**: returns true if container contains a bean definition / instance with the given name
- **`<T> T getBean(Class<T> requiredType)`**: Return the bean instance uniquely matching the given object type, if any
- **`<T> T getBean(String, Class<T> requiredType)`**: Returns a bean instance registered under the given name (type-safe)
- **`Class<?> getType(String name)`**: Returns the Class object of the bean with the given name
- **`boolean isSingleton(String)`**: Determines whether or not the named bean is a singleton
- **`String[] getAliases(String)`**: Return the aliases for the given bean name, if any
- Many more methods - covered as needed
  - View the javadoc for complete details

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

A Word About JUnit

- **JUnit** is an open source Java testing framework
  - Our Spring examples and labs use JUnit to run program code
- The JUnit 4 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
- 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:

JUnit Example

- To write a JUnit test, we will:
  - Create a class, with at least one method annotated with **`@Test`**
  - Make **assertions** using static methods in the **`org.junit.Assert`** class (e.g. **`assertTrue`**)
- Note how `springTest()` is annotated with `@Test`
  - It creates the Spring application context, and checks that it's non-null
  - We use `org.junit.Assert.`**`assertTrue`** 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 UT_Catalog {
   @Test
   public void springTest() {
      ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext();
      assertTrue("spring container should not be null", ctx != null);
   }
}
```

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());
```

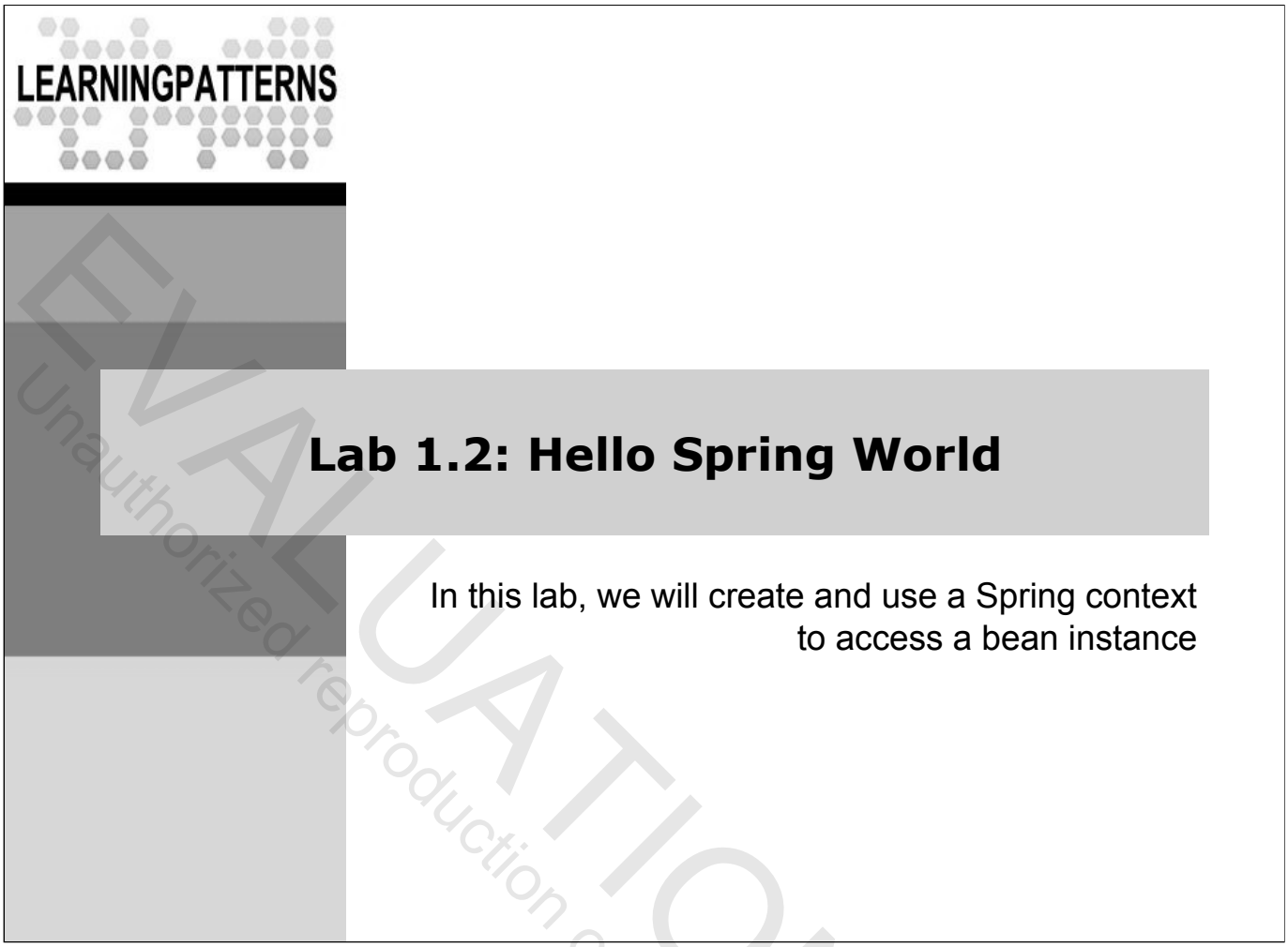
  - This is a little cumbersome, so the static import feature of Java is used - which imports static members from a class
  - The following import statement imports all the static members (including methods) from the Assert class

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

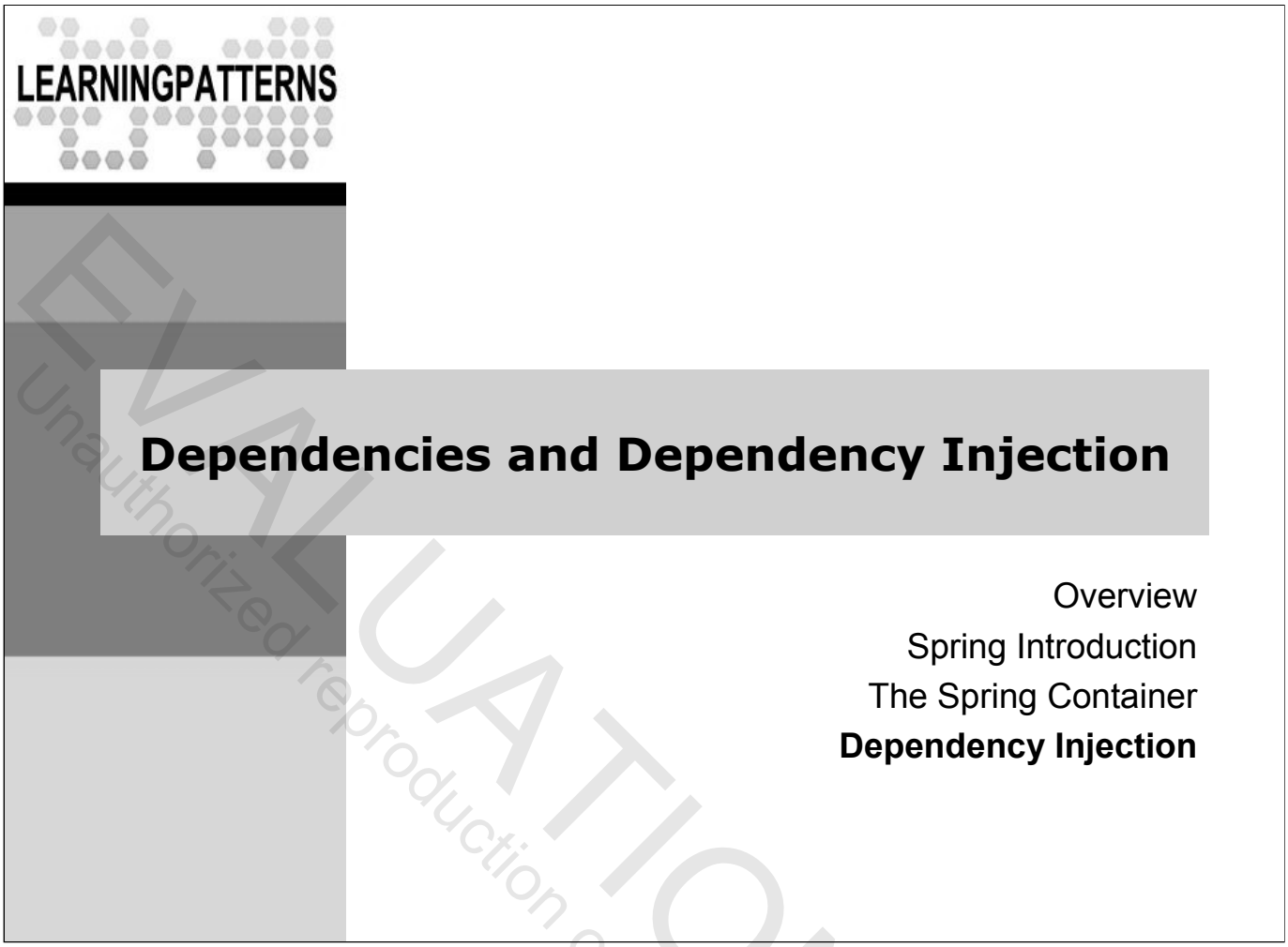
  - This allows us to use the static members without qualifying them by the classname, as seen in our earlier code

```
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



Notes:



Notes:

Dependencies Between Objects

- Multiple objects work together in an OO system
  - e.g., Object A directly uses Object B to accomplish a goal (1)
  - So Object A **depends on** Object B
- Direct dependencies can lead to unwanted traits
  - **Rigidity**: Changes affect many other parts of the system, so are hard to make
  - **Fragility**: Changes cause unexpected failures in other system areas
  - **Immobility**: Hard to reuse functionality elsewhere - modules can't be disentangled
- We'll show a direct dependency example, then show an alternative approach using **Dependency Inversion**

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

Example of a Direct Dependency

- Assume `CatalogImpl` uses `InMemoryRespository`
  - Below, `CatalogImpl` creates an `InMemoryRespository` instance directly
  - `CatalogImpl` **depends** on the lower level module details
  - To use a different data store - e.g. a `FileRespository`, then `CatalogImpl` code must change (see notes)

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

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

Notes:

- Assume that all the 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
- Assume we want to get our information from a class called `FileRespository`
  - 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 {
  FileRespository rep = new FileRespository();
  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

Dependency Inversion Principal

- High level or low level modules should **should depend upon abstractions** and not upon each other
  - **Use abstractions** (e.g. **interfaces**) that all modules depend on
  - High level modules are written in terms of the interfaces, and not directly in terms of low level modules
  - Dependencies exist, but not exposed in implementation classes
- Using this design strategy has a number of advantages:
  - Facilitates **less coupled components**
    - With a high degree of separation of responsibilities
  - Produces **greater flexibility**
    - Implementations can be swapped without affecting other modules
  - Facilitates **reuse** of components
    - They're less coupled to other parts of an application

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
- Many people don't use this design in building applications
  - Even though we know of good design principals, there is no way to make sure people use them
- 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 Java are basically reflected as dependencies between different classes

Example of Dependency Inversion

- Both `CatalogImpl` and `InMemoryItemRespository` depend on **`ItemRespository`** (an interface) – neither depend on the other

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

```
// Much detail omitted …
public class InMemoryItemRepository implements ItemRepository {
   public MusicItem get(Long id) { /* Detail omitted */  }
}
```

```
public class CatalogImpl implements Catalog {
   private ItemRepository itemRepository;
   public void setItemRepository(ItemRepository itemRepository) {
      this.itemRepository = itemRepository;
   }
   public MusicItem findById(Long id) {
      return itemRepository.get(id);
   }
}
```

Notes:

- In the code example, we can see that `CatalogImpl` knows nothing about `InMemoryItemRespository`
  - Instead, it depends on the `ItemRespository` interface (the abstraction)
  - When `CatalogImpl` is created, it is initialized with an instance of some `ItemRespository` implementation
  - However, it doesn't know any details of this implementation, and doesn't even know its exact type

- `InMemoryItemRespository` likewise depends on the abstraction (it implements the `ItemRespository` interface)
  - The abstraction (the interface) is the common language that lets the different part of the system work together with depending directly on each other

Example of Dependency Inversion

- CatalogImpl is initialized with an instance of InMemoryItemRespository (implementing ItemRespository)
  - CatalogImpl **only sees this** as the ItemRespository type
  - We could initialize CatalogImpl with any type implementing ItemRespository – e.g. a FileItemRepository
  - We've **decoupled** the modules
  - This makes them more flexible and easier to use, reuse, and maintain

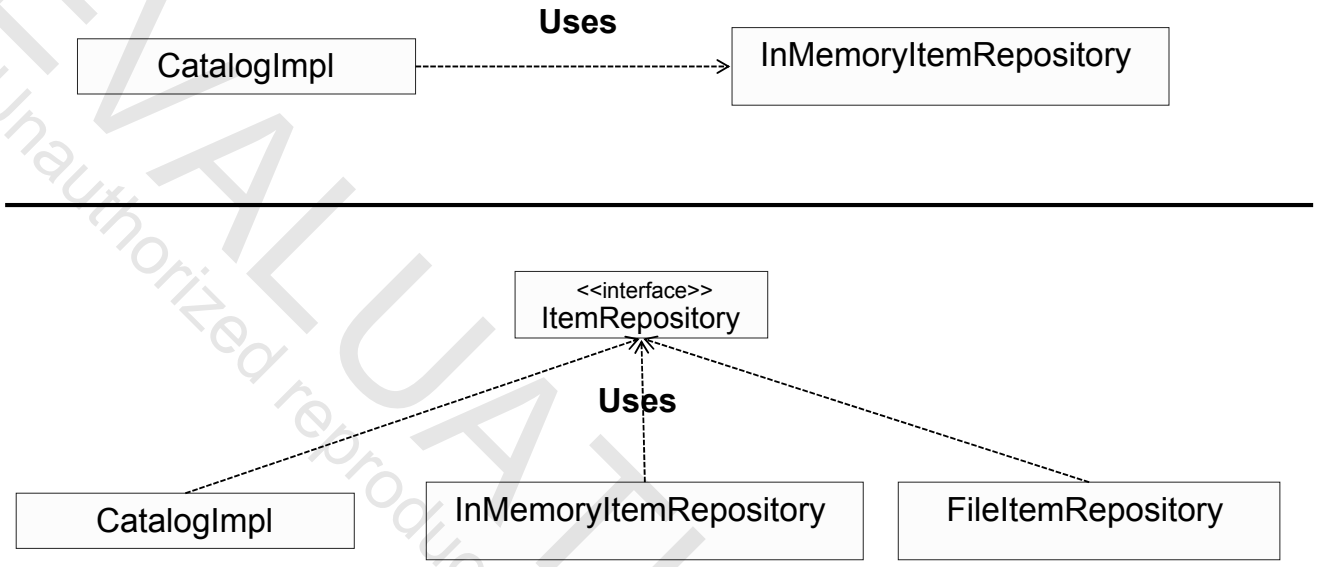
```
public class UT_Catalog {
@Test
  public void catalogTest () {
    InMemoryItemRespository rep=new InMemoryItemRespository();
    CatalogImpl catalogImpl=new CatalogImpl();
    catalogImpl.setItemRespository(rep);
    MusicItem found = catalogImpl.findById(1);
    assertTrue("item should not be null", found != null);
  }
}
```

Notes:

- In the example above, we are still creating InMemoryRespository in a fairly straightforward way
  - This exposes the actual implementation class of the lower level module (InMemoryRespository) in our UT_Catalog code
  - There are other, more sophisticated, ways to create these lower level modules
  - For example, we could use factories of some sort to more completely abstract which ItemRespository is being used
  - We will soon show how Spring can do this for you
- Whenever you have modules depending on one another, something has to know about the implementation details
  - Even if it's abstracted through multiple layers, there is still a layer somewhere which creates the instance of the actual implementation
  - The goal is to do this in a way that makes your code easier to write, reuse, and maintain
- We'll see how Spring deals with this soon

Dependency Inversion Illustrated

- Below, `CatalogImpl` depends on `InMemoryItemRepository`
- At bottom, all types depend on the `ItemRepository` abstraction



Notes:

- In the first illustration, `CatalogImpl` is directly depending on the lower level `InMemoryItemRepository`
  - `InMemoryItemRepository` is lower level because `CatalogImpl` is using `InMemoryItemRepository`
- In the bottom illustration, all modules are depending on the abstract `ItemRepository` interface
  - There is no dependency on a lower level concrete implementation
  - In fact, `CatalogImpl` is unaware of what concrete implementations exist

Dependency Injection (DI) in Spring

- **Dependency Injection** lets you abstract dependencies more
  - The Spring container **injects** dependencies into a bean
  - Done via constructors, bean properties, or factory method args
- Dependencies **are defined in the Spring configuration**
  - The **Spring container** initializes a bean's dependencies ("injects" them) based on these definitions
  - **No need to explicitly initialize dependencies** in your code
- Your bean classes remain POJOs
  - There is nothing else special about the bean classes
  - Of course, the POJO needs to be consistent with the Spring configuration (e.g. have a property to hold a dependency)

Notes:

- Dependency injection is a fancy way of saying that the container will initialize the dependencies in the bean when the bean is created
  - The bean has to be a bean whose lifecycle is managed by the container

Dependency Injection Configuration

- The (XML-based) Spring configuration below uses DI
  - Declares two beans: `itemRespository` and `musicCatalog`
  - **Injects** the `itemRespository` property of `musicCatalog`
    - Using the instance of `InMemoryItemRespository` defined before it

  `<property name="itemRespository" ref="inMemoryRespository"/>`

  - You don't need to write **any** Java code for the injection to happen

```
<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:

- The container first creates an instance of `InMemoryItemRepository`, with the id `inMemoryRepository`
- The container next creates an instance of `CatalogImpl` using the no-arg constructor
- The `<property>` element tells the container to use setter injection to initialize the dependency
  - To accomplish the DI, the container calls musicCatalog's `setItemRepository` method, passing in the instance of `InMemoryItemRepository` that it had created
  - The result is a fully initialized `CatalogImpl` instance
- The example uses "setter injection" because it's using the set method of the bean
  - This follows standard property naming conventions, so if the property is named `itemRepository`, it will call the `setItemRepository` method to initialize the dependency
  - We'll see other ways of initializing the dependency later, such as constructor injection
  - The container can also inject directly into the field (using annotation-based configuration)
- The bean reference can also be declared using a nested `<ref>` element as shown below

  `<property name="itemRepository"><ref bean="itemRepository"/></property>`
  - Generally, the ref attribute shown in the slide is easier to use and understand
  - You can use the nested element if for some reason it is clearer to express it this way than as an attribute (this is also provided for backwards compatibility with earlier Spring releases)

DI Hides Dependencies

- The example below uses our newly defined beans
  - **NO dependencies** on concrete types in the code
  - Bean lookup done in terms of the **`Catalog` interface** (1)
- Internally, `CatalogImpl` **totally decoupled** from the concrete class `InMemoryItemRespository`
  - Written in terms of interface **`ItemRespository`**
  - The concrete instance is injected **by the Spring container**
  - This is transparent to all your code

```
// Much detail omitted
@Test
public void catalogTest() {
  ClassPathXmlApplicationContext ctx =
    new ClassPathXmlApplicationContext("applicationContext.xml");
  Catalog cat = ctx.getBean(Catalog.class);  // See note (1)
  MusicItem item = cat.findById(1L);  // Use the catalog
  ctx.close();
}
```

Notes:

- (1) The bean lookup with the code 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 `UT_Catalog` code which shows that `CatalogImpl` depends on `InMemoryRepository`

  –This is defined in your configuration, and handled for you by the container
- Note that the no-arg constructor is present by default in `CatalogImpl` since we haven't defined any constructors at all

  –If we defined constructors, and still wanted to use only setter injection, as for the configuration file in this example, we would have to include a no-arg constructor

  –We'll take a look at constructor injection, which allows us to use constructor arguments, later

Advantages of Dependency Injection

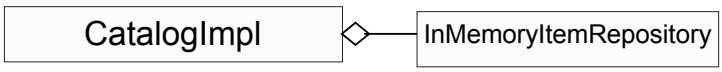
- DI **reduces coupling** between modules **in your code**
  - Coupling is basically a measure of the dependencies
- We see this in two ways:
  - `CatalogImpl` is not coupled to `InMemoryItemRespository`
  - `UT_Catalog` is not coupled to `CatalogImpl` or `InMemoryItemRespository`
- The dependencies 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:

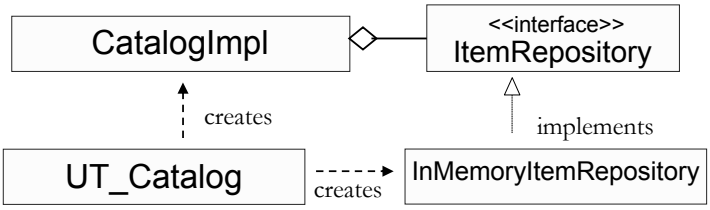
- Coupling is the measure of how much a module of code relies on other modules
  - Loosely coupled code is generally considered better code
- Consider some of the following scenarios, and how DI makes them easier
  - Testing your code with a testing framework such as JUnit can be much easier with DI – 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 the code

Dependency Injection Reduces Coupling

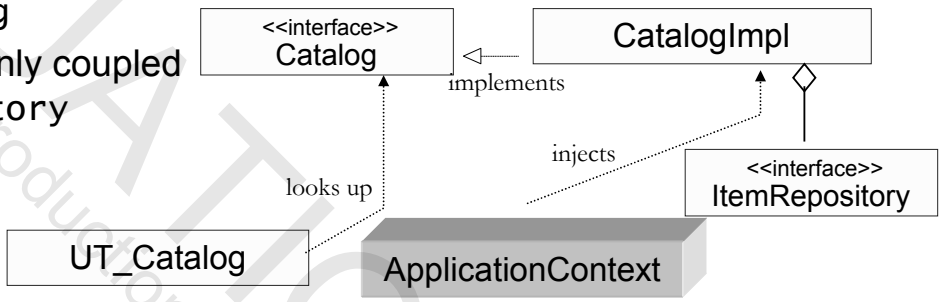
- The simplest case, `CatalogImpl` coupled to `InMemoryItemRepository`



- `CatalogImpl` coupled to `ItemRepository` only
  - `UT_Catalog` coupled to `CatalogImpl` and `InMemoryItemRepository`



- Using DI – `UT_Catalog` 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, `UT_Catalog` is doing the dependency injection of the `InMemoryItemRepository` into `CatalogImpl`
    - `UT_Catalog` is coupled to both of these types, but `CatalogImpl` is only coupled to `ItemRepository`
  - In the third, Spring is doing the DI, and so `UT_Catalog` 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