Fidelity LEAP

Technology Immersion Program

Mastering Spring and MyBatis

Introduction

Course Description

How is this course valuable to a Full Stack Engineer (FSE)?

- Spring supports software development by implementing much of the lower-level work that would otherwise be the responsibility for programmers to provide.
- One of the most important aspects of Spring is that of object (bean) creation and dependency injection. Based on configuration information, Spring will create objects and wire them together. This frees the programmer to concentrate more effort on the task of building software that fulfills the requirements of the project.
- MyBatis greatly simplifies the work of communicating with a relational database. Based on configuration information, which includes the SQL commands to execute, MyBatis takes care of the details of executing those commands.

Course Outline

Chapter 1	Introducing the Spring Framework
Chapter 2	Understanding Spring
Chapter 3	Advanced Spring Configuration
Chapter 4	Introduction to MyBatis and Spring
Chapter 5	Working Effectively with MyBatis
Chapter 6	Functional Programming

Course Objectives

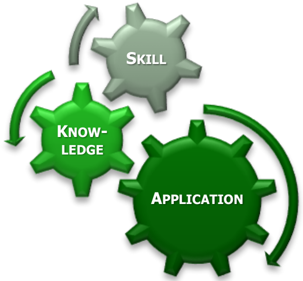
In this course, we will:

- Use the Spring framework to build clean, extensible, loosely-coupled enterprise Java applications
- Utilize Spring as an object factory and dependency injection to wire components together
- Understand and apply MyBatis to simplify access to relational databases
- Explore and apply Spring to simplify the use of MyBatis in an application
- Apply transaction strategies via configuration

Key Deliverables

Course Notes

Project Work



There is a Knowledge Checkpoint for this course

Fidelity LEAP

Technology Immersion Program

Mastering Spring and MyBatis

Chapter 1:
Introducing the Spring Framework

Chapter Overview

In this chapter, we will explore:

- The Spring Framework and its core areas of functionality
  - The Spring Object Factory
  - Inversion of Control
  - Dependency Injection

Chapter Concepts

The Spring Object Factory

XML-Based Factory Configuration

Chapter Summary

What Is Spring?

The Spring Framework is an open-source application framework and inversion of control container for the Java platform*

- The major focus of Spring is on simplifying application development
- At the core is an object factory—known as the container
- Supplemented by extensive support for application development
  - Data access
  - Web application development—MVC framework
  - Aspect-oriented programming
  - Transaction control
  - Security
  - Batch processing
  - Much more

*Source: Wikipedia

The Spring Object Factory

- Spring provides an object factory
  - Creates and manages lifecycle of application objects
  - Principle is known as *Inversion of Control* (IoC)
    - You hand-over control of object creation to Spring

- Object factory can also perform *Dependency Injection* (DI)
  - Establish links between objects it creates when dependencies exist

- Object factory needs to be configured
  - With which objects to create
  - Which dependencies to establish

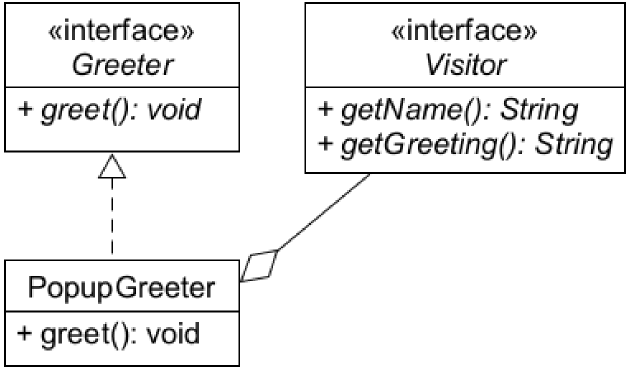
- Configuration can be performed with:
  - Annotations or
  - XML

Defining Object Dependencies

- Consider the following simple plain Java code
  - `PopupGreeter` has a dependency on `Visitor`
  - An interface is used to maintain loose coupling between the greeter and its visitor
- `visitor` field must be initialized before calling `greet()`

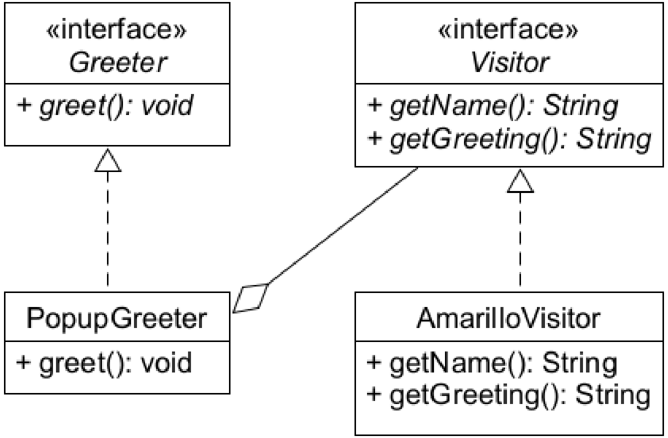
```java
public class PopupGreeter implements Greeter {
    private Visitor visitor = …;


    @Override
    public void greet() {
        String greeting = visitor.getGreeting();
        String name = visitor.getName();
        JOptionPane.showMessageDialog(null,
            greeting + ", " + name);
    }
}
```

How does `PopupGreeter` satisfy this dependency?



Defining Object Dependencies (continued)

- AmarilloVisitor is a JavaBean that implements Visitor
- Goal: satisfy PopupGreeter's dependency with an AmarilloVisitor instance



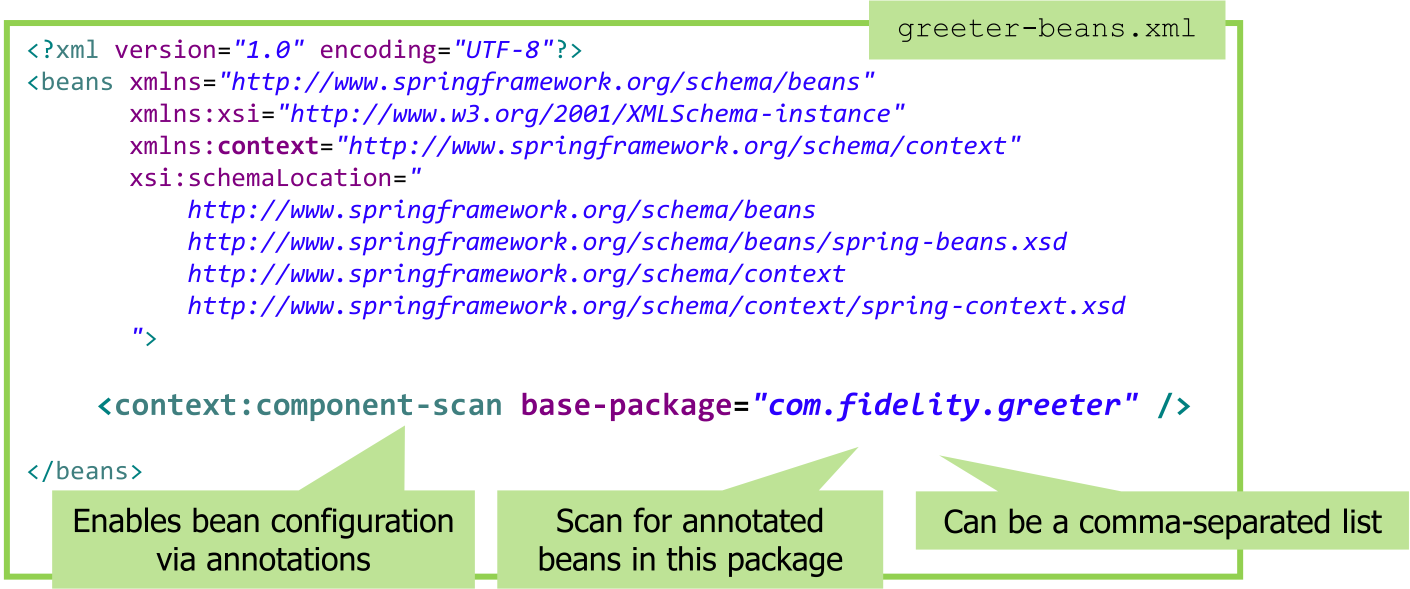
```java
public class AmarilloVisitor implements Visitor {
    private String name;
    private String greeting;

    public AmarilloVisitor() {
        this.name = "Tex";
        this.greeting = "Howdy";
    }

    @Override
    public String getGreeting() {
        return greeting;
    }

    @Override
    public String getName() {
        return name;
    }
}
```

Using Spring's Object Factory for the `PopupGreeter`

- To use the `PopupGreeter` class with the `AmarilloVisitor`, we need to:
  1. Create an instance of `PopupGreeter`
  2. Create an instance of the `AmarilloVisitor`
  3. Assign the `AmarilloVisitor` to the `PopupGreeter`'s `visitor` field
- Spring's object factory will perform all these operations for us
  - Factory will create the two objects for us – IoC
  - Factory will assign the `AmarilloVisitor` to the visitor field of `PopupGreeter` – DI
  - We never call constructors directly or manage dependencies manually
- Objects created by factory are known as *Spring-managed beans*
- To use Spring, we need to:
  1. Configure the Spring bean (object) factory
  2. Identify Spring-managed beans
  3. Instantiate the bean factory
  4. Ask the factory for the `PopupGreeter`

1. Configure the Spring Bean Factory

- In the XML configuration file, enable annotation-based bean configuration
  - `<component-scan>` element tells Spring which packages to scan for beans



greeter-beans.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       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.fidelity.greeter" />

</beans>
```

Enables bean configuration via annotations

Scan for annotated beans in this package

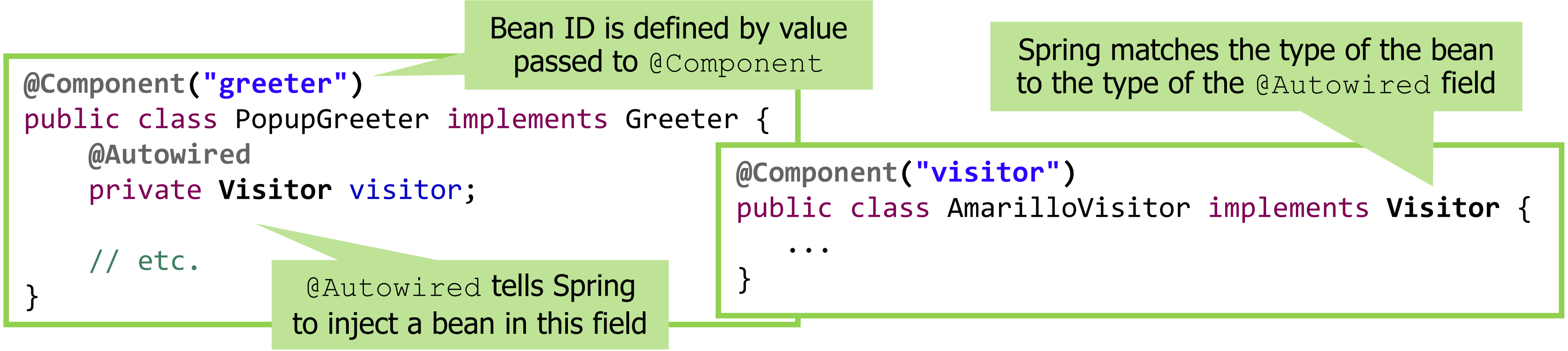
Can be a comma-separated list

2. Identify Spring-Managed Beans

- To identify your Java classes as Spring-managed beans, add annotations to them
- `@Component` is Spring's primary bean annotation
- To identify fields that require dependency injection, add annotations
  - `@Autowired` – Spring-specific
  - `@Inject` – Java EE standard annotation
    - Requires Java EE JAR file in classpath
  - Spring supports DI using either annotation

Defining Spring-Managed Beans

- Annotate Spring-managed beans with `@Component`



Bean ID is defined by value passed to `@Component`

```
@Component("greeter")
public class PopupGreeter implements Greeter {
    @Autowired
    private Visitor visitor;

    // etc.
}
```

`@Autowired` tells Spring to inject a bean in this field

Spring matches the type of the bean to the type of the `@Autowired` field

```
@Component("visitor")
public class AmarilloVisitor implements Visitor {
    ...
}
```

- When Spring finds a bean with `@Component`, it calls its constructor
  - Then it looks for `@Component` beans whose types match fields with `@Autowired`
  - When it finds matching beans, it injects them into the `@Autowired` fields
  - This process continues recursively until all dependencies for all beans are satisfied

Spring Annotation for Dependency Injection

- `@Autowired` tells Spring where DI is needed
- `@Autowired` can be added to:
  - Fields

```
@Autowired
private Visitor visitor;
```

Inject bean by assigning it directly to a field

  - Setter methods

```
@Autowired
public setVisitor(Visitor v) { … }
```

Inject bean by passing it as an argument to a setter method

  - Constructors

```
public class PopupGreeter {
    @Autowired
    public PopupGreeter(Visitor v) { … }
```

Inject bean by passing it as an argument to the constructor

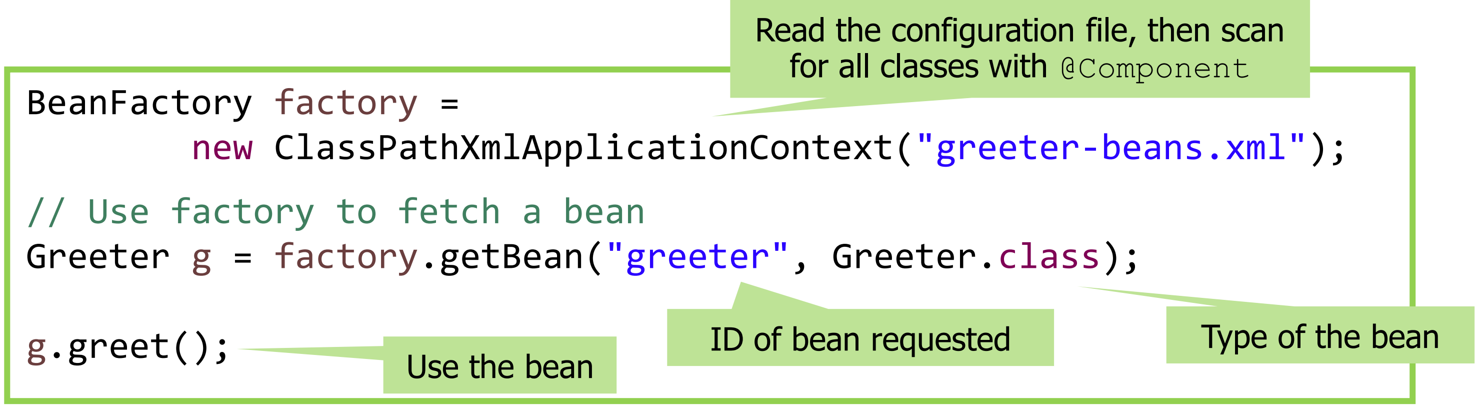
- Spring will create and inject an object of the right type
  - No problem as long as **only one** Spring-managed bean implements `Visitor`

Creating Beans with Bean Factories

- Spring provides several factories for managing beans
  - All of them implement the `BeanFactory` interface
    - `org.springframework.beans.factory.BeanFactory`
  - Configuration for the factory is provided with Java annotations or XML files or both
- Any `BeanFactory` implementation is capable of:
  - Instantiating beans (IoC)
  - Injecting the bean's dependencies (DI)
  - Providing access to those beans
- Many Spring applications often define the factory as `ApplicationContext`
  - `org.springframework.context.ApplicationContext`
  - A subinterface of `BeanFactory` that adds extra functionality
- Each factory is a bean "container"
  - Manages the lifecycle of the beans that it creates

3. Instantiate the Bean Factory, 4. Get Beans

- Instantiate a bean factory by calling a factory constructor
  - `ClassPathXmlApplicationContext` reads configuration from XML file on classpath
  - Bean factory creates all beans as soon as it reads its configuration (eager instantiation)
- Fetch beans from a factory using its `getBean()` method
  - Requires a bean ID and the bean's type (class or interface)



```java
BeanFactory factory =
        new ClassPathXmlApplicationContext("greeter-beans.xml");

// Use factory to fetch a bean
Greeter g = factory.getBean("greeter", Greeter.class);

g.greet();
```

Read the configuration file, then scan for all classes with `@Component`

ID of bean requested

Type of the bean

Use the bean

AbstractApplicationContext

- For applications deployed in an application server, the bean container shuts down automatically
  - But standalone clients (such as our lab exercises) must close the container explicitly
- `BeanFactory` interface doesn’t define a `close()` method
  - So, we declare the factory as `AbstractApplicationContext`
  - Best practice: use the most general type that provides the required functionality

```
AbstractApplicationContext factory =
                new ClassPathXmlApplicationContext("greeter-beans.xml");
…
factory.close();
```

Bean factory will clean up all beans that it is managing

Other Spring Annotations for Defining Beans

- `@Component` has subinterfaces for beans with specialized roles:
  - `@Controller` – bean associated with the presentation tier or a REST service endpoint
  - `@Service` – bean associated with the business tier (business logic)
  - `@Repository` – bean associated with the integration tier (DAO)

- By default, beans are *singletons*
  - Multiple calls to `factory.getBean()` return a reference to the same bean

- Change the bean's scope with `@Scope`
  - `prototype` – each call to `factory.getBean()` returns a reference to a new bean

```
@Component("paymentProcessor")
@Scope("prototype")
public class CreditCardPaymentProcessor implements PaymentProcessor {
```

Default scope is `singleton`

`@Qualifier` and `@Primary`

- For DI, there must be only one bean of the required type; otherwise, an exception occurs
  – If there are multiple beans of that type, add `@Qualifier` with a bean ID

```
@Component("greeter")
public class PopupGreeter … {
    @Autowired
    @Qualifier("dehliVis")
    private Visitor visitor;

    …
```

Inject the `Visitor` with ID `dehliVis`

```
@Component("delhiVis")
public class DehliVisitor implements Visitor {

    …
```

```
@Component("dublinVis")
public class DublinVisitor
    implements Visitor { … }
```

- Or annotate one of the bean classes with `@Primary`

```
@Component("greeter")
public class PopupGreeter … {
    @Autowired
    private Visitor visitor;

    …
```

No `@Qualifier` needed

Use this bean if there are multiple `Visitor` implementations

```
@Primary
@Component("delhiVis")
public class DehliVisitor implements Visitor {

    …
```

Exercise 1.1: Spring with Annotations

HANDS-ON EXERCISE

30 min

- Follow the directions in your Exercise Manual

Chapter Concepts

The Spring Object Factory

XML-Based Factory Configuration

Chapter Summary

XML Configuration File

- Instead of using annotations, you can define Spring beans in the XML configuration file
- Configure a bean with the `<bean>` element's attributes
  - `id` – sets the bean's ID
  - `class` – fully-qualified class name so Spring knows which constructor to call

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">


    <bean id="greeter" class="com.fidelity.greeter.PopupGreeter" />

    <bean id="vis" class="com.fidelity.greeter.AmarilloVisitor" />

</beans>
```

Bean ID passed to factory's `getBean()`

Configuring Dependency Injection in XML

- To configure DI using JavaBean setter methods, add `<property>` elements to a bean
  - `name=`*`property`* – tells Spring which setter method to call: `set`*`Property`*`(…)`

- Define setter method arguments using `<property>` element attributes
  - `ref` – reference to another Spring bean
  - `value` – literal string or number value

```
<beans …>
    <bean id="greeter"
          class="com.fidelity.greeter.PopupGreeter">
       <property name="visitor" ref="vis"/>
    </bean>
```

```
public class PopupGreeter implements Greeter {
  public void setVisitor(Visitor visitor) { … }
```

Call `setVisitor()`…

… with the bean whose ID is `vis` as the argument

```
    <bean id="vis"
          class="com.fidelity.greeter.AmarilloVisitor">
       <property name="name" value="Joe Bob Springsteen"/>
    </bean>
```

```
public class AmarilloVisitor implements Visitor {
  public void setName(String name) { … }
```

Call `setName()`…

… with this string value as the argument

Using Beans Defined with XML from Factory

- Get beans defined using XML exactly as with beans defined using annotations

Read bean configuration file

```
BeanFactory factory =
        new ClassPathXmlApplicationContext("greeter-beans.xml");

// Use factory
Greeter g = factory.getBean("greeter", Greeter.class);

g.greet();
```

Specify value of `<bean>` element's `id` attribute

Use the bean

Exercise 1.2: Spring with XML

HANDS-ON EXERCISE

30 min

- Follow the directions in your Exercise Manual

How Many Beans Will Be Created?

- The `getBean()` method may:

1. Keep returning the same object each time
   - A “singleton”
   - One bean per Spring container (`BeanFactory` instance)
2. Or return a freshly instantiated object each time
   - Uses the bean configuration as a “prototype”
   - Return a new object for every distinct request made to the factory

- The number of beans created is controlled by the *scope* configuration of the bean
  - Available scopes:
    - Singleton and prototype for all Spring applications
    - Request and session for web applications

We will use only these two scopes

Bean Scope Example

- Define the scope of a bean with the `<bean>` element's `scope` attribute
  – If omitted, `singleton` is the default

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

    <bean id="greeter" class="com.fidelity.greeter.PopupGreeter" scope="prototype">
        <property name="visitor" ref="vis"/>
    </bean>

    <bean id="vis" class="com.fidelity.greeter.AmarilloVisitor">
        <property name="name" value="Joe Bob Springsteen"/>
    </bean>
</beans>
```

Each call to `getBean()` creates a new object

Default `singleton` scope

Complete Configuration Using Annotations

- You can eliminate the XML configuration file completely using Java Configuration
- Define a class with `@Configuration`
  - Define factory methods with `@Bean`
  - When Spring needs to inject a bean, it examines the return types of your `@Bean` methods
  - Spring calls the appropriate `@Bean` method to create the bean for injection
- Classes of objects returned by `@Bean` methods don't need `@Component`
  - Useful for creating beans from classes you don't own

```
@Configuration
public class AppConfig {

    @Bean
    public Visitor createVisitor(){
        return new AmarilloVisitor();
    }

    @Bean(name="greeter")
    public Greeter createGreeter(){
        return new PopupGreeter();
    }
}
```

Annotation-Based Factory Creation

- Use your configuration class to create a `BeanFactory`
  - Call `AnnotationConfigApplicationContext` constructor
  - Pass a reference to your configuration class as the constructor argument

Create a bean factory

Your configuration class

```
BeanFactory factory =
        new AnnotationConfigApplicationContext(AppConfig.class);

Greeter g = factory.getBean("greeter", Greeter.class);

System.out.println("Got greeter " + g);

g.greet();
```

Fetch beans as usual

Use Case for Java Configuration: Injecting a Logger

- Use Java Configuration to create beans from classes that aren't Spring beans
  - Example: injecting an SLF4J `Logger` instance

BEFORE: `Logger` isn't a Spring bean, so Spring can't inject it

```
@Component("greeter") public class PopupGreeter … {
    private Logger logger = LoggerFactory.getLogger(getClass());
```

```
@Configuration
public class MyJavaConfig {
    @Bean
    @Scope("prototype")
    public Logger createLogger(InjectionPoint ip) {
        Class<?> classThatWantsALogger = ip.getField().getDeclaringClass();
        return LoggerFactory.getLogger(classThatWantsALogger);
    }
```

Spring will call this method when it needs a `Logger` instance

Spring automatically performs injection for the `@Bean` method's argument

Create a `Logger`

```
@Component("greeter") public class PopupGreeter … {
    @Autowired
    private Logger logger;
```

AFTER: Spring calls our `@Bean` method and injects a `Logger`

Exercise 1.3: Java Configuration

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Concepts

The Spring Object Factory

XML-Based Factory Configuration

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The Spring Framework and its core areas of functionality
  - The Spring Object Factory
  - Inversion of Control
  - Dependency Injection

Key Points

- Spring provides a general-purpose object factory
  - Factory performs dependency injection when configured to do so

- To use the Spring factory:

1. Configure the factory using annotations or XML or both
2. Create an `ApplicationContext`
3. Get beans from the `ApplicationContext`
   - Using the id of the bean and its interface type

Fidelity LEAP

Technology Immersion Program

Mastering Spring and MyBatis

Chapter 2: Understanding Spring

Chapter Overview

In this chapter, we will explore:

- Spring’s dependency injection
- Testing with Spring dependency injection
- Working with Maps

Chapter Concepts

Spring and Dependency Injection

Testing with Spring

Working with Maps

Other Dependency Types

Chapter Summary

Spring Dependency Injection

déjà vu

Earlier, we introduced Spring:

- Spring provides an object factory
  - Creates and manages lifecycle of application objects
  - Principle is known as *Inversion of Control* (IoC)
    - You hand-over control of object creation to Spring
- Object factory can also perform *Dependency Injection* (DI)
  - Establish links between objects it creates when dependencies exist
- These features will be illustrated on the following slides

Injecting Beans Using Annotations

- Identify Spring-managed Beans using `@Component`
  - Or the more specific `@Controller`, `@Service`, and `@Repository`

- Specify that a dependency needs to be injected
  - `@Autowired` can be on fields, methods, or constructor arguments

- Spring will inject an object of the right type
  - No problem as long as only one Spring-managed component implements the interface
  - Add `@Qualifier` or `@Primary` if there could be multiple beans of required type

- Can also configure Spring *profiles* for defining different beans in different environments
  - Example: different DAO beans for development, test, and production environments
  - We'll discuss profiles later

Setter Injection – What Does Spring Do?

- It is helpful to look at a Spring configuration file and think about the equivalent code that Spring is "writing" behind the scenes
  - Example: how does Spring handle the following XML configuration?

```xml
<bean id="vis" class="com.fidelity.greeter.AmarilloVisitor" />

<bean id="greeter" class="com.fidelity.greeter.PopupGreeter">
    <property name="visitor" ref="vis"/>
</bean>
```

Triggers call to `setVisitor()`

- This is the code Spring executes:

Call constructor to create bean

```java
Visitor vis = new com.fidelity.greeter.AmarilloVisitor();

Greeter g = new com.fidelity.greeter.PopupGreeter();
g.setVisitor(vis);
```

Call constructor to create dependent bean

Inject dependency

Injecting a Simple Value

- IoC container can inject many types of *values*
  - Not just other beans
  - Although beans are the most common
- Can inject primitives and Strings ("values"):

```java
public class AmarilloVisitor implements Visitor {
    private String name;
    …
    public void setName(String name) {
        this.name = name;
    }
}
```

```xml
<bean id="vis" class="com.fidelity.greeter.AmarilloVisitor">
    <property name="name" value="Joe Bob Springstein"/>
</bean>
```

- Value injection is available with annotations
  - Here, it's silly: just assign the value instead
  - Later, we'll see useful examples with the Spring Expression Language (SpEL)

```java
public class AmarilloVisitor implements Visitor {
    @Value(value="Joe Bob Springstein")
    private String name;
    …
}
```

Exercise 2.1: Using Spring as a Factory

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Invoking the Default Constructor

- When beans are defined like this:

```
@Component("greeter")
public class PopupGreeter implements Greeter {
    @Autowired
    private Visitor visitor;

    public PopupGreeter() {}
    …
```

```
@Component("visitor")
public class DublinVisitor implements Visitor {

    public DublinVisitor() {}
    …
```

  – The BeanFactory uses the default zero-argument constructors to create objects

- The `DublinVisitor` and `PopupGreeter` classes must have zero-argument constructors
  – The fields are set with field injection, not in the constructor

Defining Beans with Constructor Arguments

- The `@Autowired` annotation also supports constructors

```
public class PopupGreeter implements Greeter {
    private Visitor visitor;
    @Autowired
    public PopupGreeter(Visitor visitor) {
        this.visitor = visitor;
    }
…
```

No zero-arg constructor

Spring creates a `Visitor` first, then calls the `PopupGreeter` constructor

- Can apply `@Qualifier` to the constructor parameters
  – To handle parameter conflict or multiple beans with same interface

```
public class PopupGreeter implements Greeter {
    private Visitor visitor;
    @Autowired
    public PopupGreeter(@Qualifier("amarilloVis") Visitor visitor) {
        this.visitor = visitor;
    }
…
```

Configuring Constructor Arguments with XML

- You can also configure constructor arguments in XML
  - Use `<constructor-arg>` element

```xml
<bean id="star"
      class="com.fidelity.dependency.Actor">

   <constructor-arg value="Tom Hanks" />
   <constructor-arg value="192000" />

</bean>
```

```java
public class Actor {
    private String name;
    private String dailyRateUsd;

    public Actor(String name, int dailyRateUsd) {
        this.name = name;
        this.dailyRateUsd = dailyRateUsd;
    }
```

- `<constructor-arg>` has attributes `ref` and `value` attributes like `<property>`

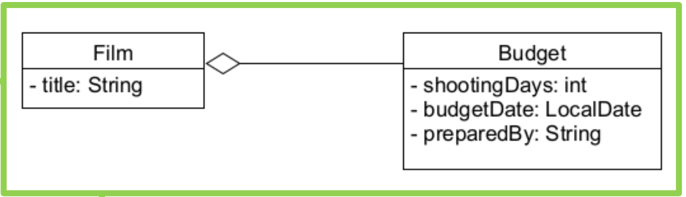
- If the constructor has multiple arguments, Spring will attempt to match by type
  - Even if `<constructor-arg>` elements are out of order
  - To avoid potentially ambiguous calls, specify the order with the `index` attribute

```xml
<bean id="star" class="com.fidelity.dependency.Actor">
    <constructor-arg index="1" value="192000" />
    <constructor-arg index="0" value="Tom Hanks" />
</bean>
```

No problem even if arguments are out of order

Fully Configured Beans

- Spring will not inject a bean unless the bean is completely configured
  - Example: `Film` bean has a dependency on `Budget` bean



```java
@Component
public class Film {
    private Budget budget;

    @Autowired
    public Film(Budget budget) {
        this.budget = budget;
    }
}
```

```java
@Component
public class Budget {
    // no reference to a Film
}
```

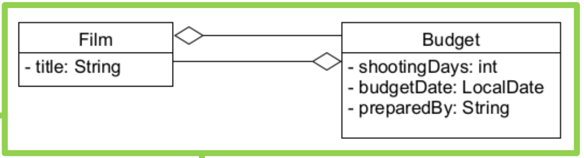
- Spring analyzes dependencies between beans and creates them in the correct order
  - Here, Spring creates the `Budget` bean first, then passes it to the `Film` constructor

Circular Dependency

- But what if `Budget` requires a reference to the `Film`?

```
@Component
public class Film {
    private Budget budget;

    @Autowired
    public Film(Budget budget) {
        this.budget = budget;
    }
}
```

```
@Component
public class Budget {
    private Film film;

    @Autowired
    public Budget(Film film) {
        this.film = film;
    }
}
```



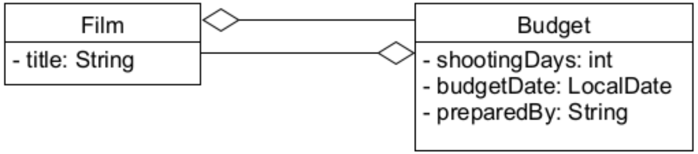
- Can't create `Film` first because its constructor needs a fully configured `Budget`
  - Can't create `Budget` first because it needs a `Film`
- Problem: circular dependency
  - Spring will throw exception (`BeanCurrentlyInCreationException`)

Resolving Circular Dependencies

- To avoid circular dependencies:
  – Use setter injection or field injection for at least one link in the chain

```
@Component
public class Film {
    private Budget budget;


    @Autowired
    public Film(Budget budget) {
        this.budget = budget;
    }
}
```

Constructor injection



```
@Component
public class Budget {
    @Autowired
    private Film film;

    public Budget() {
    }
}
```

Field injection

`Budget` constructor doesn't need a `Film`

- Spring will create the `Budget` bean first, but will postpone autowiring its `Film` field
  – Then Spring passes the `Budget` bean to the `Film` constructor
  – Finally, Spring injects the initialized `Film` bean into the `Budget` field

Constructor or Setter Injection?

- When designing beans, you have three choices
  1. Add `@Autowired` to constructors
  2. Add `@Autowired` to setter methods
  3. Add `@Autowired` to fields

- Which one should you choose?

- In early releases of Spring, the Spring team recommended using setter injection exclusively
  - More flexible, prevents circular dependencies
  - Disadvantage: developer may forget to set a required dependency

- The Spring team now recommends using constructor injection
  - Guarantees all beans are completely initialized
  - Disadvantage of constructor injection: potential for circular dependencies

When to Use Constructors and Setters

- Use constructor injection for:
  - “Read-only” objects
    - Should not be changed after construction
    - Avoid setters in objects that should be immutable
  - Dependencies that are required for bean to function properly
    - Avoid runtime errors due to misconfiguration

- Use setter or field injection for:
  - Cases where a circular dependency results
  - Mutable objects and dependencies
  - Optional dependencies

- Decide on a dependency-by-dependency basis
  - Some fields can be initialized in a constructor, some in fields, others with setters

Exercise 2.2: Dependency Injection with Constructors

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Concepts

Spring and Dependency Injection

Testing with Spring

Working with Maps

Other Dependency Types

Chapter Summary

Unit Testing with Spring

- Best practice is to satisfy dependencies manually
  - Instantiate objects with `new`, or
  - Create mock dependencies with Mockito and inject with constructor or setters

- Spring supplies test support classes to help mimic “Spring-like” behavior in unit tests
  - `ReflectionTestUtils` can set fields or invoke methods (even if `private`)

```java
@Service
public class BusinessService {
    @Autowired
    private ProductDao dao;

    …
```

BusinessServiceTest.java

```java
class BusinessServiceTest {
   BusinessService service;

   @BeforeEach
   void setUp() {
      ProductDao testDao = new ProductDaoImpl();
      service = new BusinessService();
      ReflectionTestUtils.setField(service, "dao", testDao);
   }
```

Set the field named `dao` in the `testService` object

- Another option is to supply a special testing configuration and use the Spring bean factory
  - More suited to integration testing, but sometimes used for unit testing

Integration Testing Spring with `@ExtendWith`

- Goal: define integration tests of Spring with our application
  - Purpose: test the Spring configuration
- Spring integration testing setup:

1. Annotate the test class with `@ExtendWith(SpringExtension.class)`
2. Annotate the test class with the `@ContextConfiguration`
   - Pass in `beans.xml` file
   - If using a `@Configuration` class, pass it in as the `classes` parameter
3. Declare beans being tested (typically Service or DAO) as `@Autowired` fields

- In the `@Test` methods, all beans will be fully configured by Spring
  - Autowired beans will be fully initialized
  - All dependencies are injected recursively

Using the Spring TestContext Framework

- Instead of this:

```
class BusinessServiceGetBeanTest {
    BusinessService service;
    AbstractApplicationContext context;

    @BeforeEach
    void setUp() {
        context = new ClassPathXmlApplicationContext("beans.xml");
        service = context.getBean(BusinessService.class);
    }

    @AfterEach
    void tearDown() {
        context.close();
    }
```

- Use this:

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration("classpath:beans.xml")
class BusinessServiceSpringTest {
    @Autowired
    BusinessService service;
```

Or `(classes = ApplicationConfig.class)`

Exercise 2.3: Integration Testing with Spring

Hands-On Exercise

10 min

- Follow the directions in your Exercise Manual

Chapter Concepts

Spring and Dependency Injection

Testing with Spring

Working with Maps

Other Dependency Types

Chapter Summary

Overview of Collection Framework Interfaces

déjà vu

- Set
  - Holds onto unique values
  - Can be used to check for existence of objects
- List
  - Like arrays, only can grow and shrink
- Queue and Deque
  - Used to store items for processing, add and remove methods
  - Deque allows to add or remove from front and back of container
- Maps
  - Stores key/value pairs
  - Helpful to cache infrequently changed data from files or database

HashMap

- Each `Entry` in a `Map` is a pair
  - Key
  - Value
- `HashMap` is the most commonly-used map
- `TreeMap` stores data in sorted order
- Useful methods of `Map`:
  - `put()` allows you to add or replace items in the map
  - `get()` allows you to obtain items from the map
    - A search operation
- `Maps` are commonly used to cache data
  - The result of database queries
  - Content of files that may be read multiple times

Using `HashMap`: An Example

- Example: `Employee` object is the key; `Phone` object is the value

```
// set up query etc

Map<Employee, Phone> directory = new HashMap<>();
try (PreparedStatement stmt = conn.prepareStatement(sql)) {
    …
    ResultSet rs = stmt.executeQuery();
    while (rs.next()) {
        // Create an Employee using rs.getXXXXX()
        Employee emp = new Employee(…);
        // Create a Phone by the same mechanism
        Phone phone = new Phone(…);
        directory.put(emp, phone);
    }
}
// etc
```

```
// elsewhere
Employee boss = …;
Phone bossPhone = directory.get(boss);
```

What Kinds of Objects Are Valid Keys to a Map?

- Both Key and Value can be any Object
  - `String`, `Employee`, `Phone`, `Double`, `Integer`, etc.
  - Not primitives such as `int`, `char`
    - Use the corresponding wrapper class: `Integer`, `Char`
- Keys must not be null
- For `HashMap`, the Key class must override `hashCode()` and `equals()`
  - The hash code is usually computed from all the fields of an object
- For `TreeMap`, Key class must implement `Comparable`
  - Otherwise, when you create a `TreeMap` you supply a `Comparator` that compares Keys

Converting Map to Other Collections

- In a `Map<K, V>`
  - Sometimes we want to work with a whole entry (key and value)
    - Can use `Map.Entry<K, V>`
- `keySet()`
  - Returns `Set<K>` containing all the keys (keys must be unique, so this is appropriate)
- `values()`
  - Returns `Collection<V>` containing all the values in the map
  - Usually processed as a `List`, since values may not be unique
- `entrySet()`
  - Returns `Set<Map.Entry<K,V>>` containing the mappings in the map

Exercise 2.4: Create and Access a `HashMap`

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Concepts

Spring and Dependency Injection

Testing with Spring

Working with Maps

Other Dependency Types

Chapter Summary

Injecting Lists and Sets

- Can inject a `java.util.List` of values with XML configuration
  - If the Java code uses generics, Spring will ensure type safety

```xml
<bean id="no-way" class="com.fidelity.dependency.Film">
    <property name="cast">
        <list>
            <ref bean="chopra" />
            <ref bean="hanks" />
            <ref bean="jing" />
        </list>
    </property>
</bean>
```

```java
public void setCast(List<Actor> cast) {
    this.cast = cast;
}
```

  - Can also put `value` elements in the list
- Similarly, the `<set>` element works for `java.util.Set`
- No precise equivalent in annotations

Using Properties in `@Configuration`

- You can define property values in an external property file
  – Read properties using `@PropertySource` on your `@Configuration` bean

```java
@Configuration
@PropertySource("classpath:film.properties")
public class AppConfig {
}
```

film.properties

```
title = It's A Wonderful Life
```

```java
@Component
public class Film {
    private List<Actor> cast;
    private String title;

    // Set value from properties file
    @Value("${title}")
    public void setTitle(String title) {
        this.title = title;
    }
    …
}
```

Property placeholder

Using Properties in `beans.xml`

- Beans defined in XML can also define read external properties file
  - Avoids need to edit `beans.xml` when changing the values

```
<beans xmlns:context="http://www.springframework.org/schema/context"
   … >
   <!-- Read the database credentials from the db.properties file -->
   <context:property-placeholder location="classpath:db.properties" />

   <!-- Define a DataSource for the database -->
   <bean id="oracleDataSource"
         class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="url" value="${db.url}" />
    <property name="driverClassName" value="${db.driver}" />
    <property name="username" value="${db.username}" />
    <property name="password" value="${db.password}" />
</bean>
```

Load the property file

Use property placeholders to use values

```
db.url = jdbc:oracle:thin:@localhost:1521:xepdb1
db.driver = oracle.jdbc.driver.OracleDriver
…
```

`db.properties`

Chapter Concepts

Spring and Dependency Injection

Testing with Spring

Working with Maps

Other Dependency Types

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Spring’s dependency injection
- Testing with Spring dependency injection
- Working with Maps

Key Points

- Spring provides a BeanFactory that supports dependency injection
- Spring can use either setters or constructors for dependency injection
- When testing with Spring, you can:
  - Satisfy dependencies manually
  - Use the Spring TestContext Framework
- Maps can be used to store values that are retrieved by providing a key
- Dependencies can be:
  - Other beans: use “ref”
  - Can be values (String, primitives): use “value”
  - Can be list, set, properties

Fidelity LEAP

Technology Immersion Program

Mastering Spring and MyBatis

Chapter 3:
Advanced Spring Configuration

Chapter Overview

In this chapter, we will explore:

- The bean lifecycle
- How dependencies are injected
- Dynamically evaluating expressions with Spring Expression Language (SpEL)
- Managing Spring configuration across multiple files
- Debugging Spring configuration problems

Chapter Concepts

Managing the Bean Lifecycle

Spring Expression Language (SpEL)

More Configuration Options

Debugging Spring Configuration Problems

Chapter Summary

`PostConstruct` and `PreDestroy`

- In our JDBC tests, we used the `tearDown()` method to call `close()` on the DAO
  - What about in a real-world application?

- When you configure a bean:
  - Can specify a pre-destroy method
    - Called only for singleton beans
    - To dispose of resources the bean itself allocated
  - Can specify a post-construction method
    - If bean requires additional initialization after all dependencies are injected

```
import javax.annotation.*;
// Java EE 9+: jakarta.annotation.*;

@Repository("dao")
public class DepartmentDao {

    @PostConstruct
    public void init() {
        …
    }

    @PreDestroy
    public void close() {
        …
    }
```

Called after all dependencies are injected

Called when Spring container shuts down

Specifying an Initialization and Destroy Method in XML

- Can specify `init-method` and `destroy-method` using XML configuration:

```
<bean id="dao"
      class="com.fidelity.advanced.DepartmentDao"
      init-method="init"
      destroy-method="close" >
    …
</bean>
```

Ensuring Graceful Shutdown

- For desktop Java applications:
  - The `ApplicationContext` needs to register a shutdown hook with JVM
  - The method is defined on the `AbstractApplicationContext` interface

```
AbstractApplicationContext factory =
        new ClassPathXmlApplicationContext(springConfigurationFile);

factory.registerShutdownHook();
```

Ensure factory is automatically closed when JVM shuts down

- This is not needed in Java EE environments (e.g., web applications and RESTful services)
  - A Spring container running in a Java EE server registers its own shutdown hook

Optional Exercise 3.1: Bean Destroy

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Concepts

Managing the Bean Lifecycle

Spring Expression Language (SpEL)

More Configuration Options

Debugging Spring Configuration Problems

Chapter Summary

Spring Expression Language

- Use the Spring Expression Language (SpEL) to execute Java code in `@Value()`
  - Syntax: `#{ `*`java-expression`*` }`
  - Note: use `#{…}` for SpEL expressions, use `${…}` for values in `.properties` files
- Example: injecting environment variables and Java system properties into beans

```
@Value("#{systemProperties['user.country']}")
private String country;
@Value("#{environment['SystemRoot']}")
private String rootDir;
```

`systemProperties` and `environment` are pre-defined SpEL beans

- Set system properties with **`java -D`***`property-name`***`=`***`property-value`*

```
$  java -Duser.country=India ...
```

- Set environment variables on the command line:
  - UNIX/Linux:

```
$  export SystemRoot=/local
```

  - MS Windows:

```
>  set SystemRoot=D:\
```

Calling Methods and Performing Calculations in SpEL

- An SpEL expression can include complex expressions
- Example: convert the value of a system property to upper case

```
@Value("#{systemProperties['currency.code'].toUpperCase()}")
private String currencyCode;
```

- Example: convert the value of the `tax_percent` environment variable to a `double`
  - Then divide by 100 to yield a tax rate
  - To reference a class in the SpEL expression, wrap it in the `T()` (type) function

```
$  export tax_percent=7
```

```
@Value("#{T(java.lang.Double).valueOf(environment['tax_percent']) / 100.0}")
private double taxRate;  // 0.07
```

Chapter Concepts

Managing the Bean Lifecycle

Spring Expression Language (SpEL)

More Configuration Options

Debugging Spring Configuration Problems

Chapter Summary

Motivation for Modularizing Spring Configuration

- Putting all Spring configuration in a single file or class can become unmanageable
  - Difficult to find individual beans
  - Becomes a bottleneck in development
- The solution is to divide the configuration according to project-specific criteria
  - E.g., per feature, per layer
- Example: when using the Spring TestContext Framework, may want a special configuration
  - Different dependencies (e.g., replace production dependencies with mock objects)
  - Do not want to duplicate unchanged dependencies
  - Do not want to “pollute” production configuration with test
- Everywhere that accepts a configuration file or class argument also accepts an array or list
  - In Java, parameter is varargs, meaning a comma-separated list or an array
  - In annotations, an array (use array constructor {…})
  - In XML, can be comma-, space-, or semicolon-separated

Importing Configuration

- Usually prefer to have more control than a simple list can provide
  - Create a single “entry point” configuration for each situation and import the rest

```
…
<import resource="classpath:common-beans.xml" />

<bean id="mockStringProvider" class="com.fidelity.services.StringProviderMockImpl" />
…
```

  - Or, using annotations

```
@Configuration
@Import(ApplicationCommonConfig.class)
public class ApplicationTestConfig {
    @Bean
    public StringProvider mockStringProvider() {
        return new StringProviderMockImpl();
    }
…
```

XML and Annotations

- It is possible to configure some beans in Java and some in XML
  - Could even have some properties in XML and others with annotations
- In the case of a conflict, XML configuration takes precedence
- Can even combine `@Configuration` classes and XML
  - If using an XML context, must have `component-scan`
    - Any classes annotated with `@Configuration` will be found and processed
    - Or can be defined with `<bean>`
  - If using an Annotation context, can import XML

```
@Configuration
@ImportResource("classpath:mixed2-beans.xml")
public class AppConfig {
```

- `@Autowired` annotations are processed on `@Configuration` classes
  - Processed very early, limit use to simple dependencies

@ComponentScan

- Can also annotate the class with `@ComponentScan`
  - Will scan current package and all sub-packages for Spring beans

```
@Configuration
@ComponentScan
public class ApplicationConfig {
    …
```

- Can also supply a list of packages in Strings
  - Or a list of classes
    - Spring will scan the packages of those classes
    - More type-safe since the compiler can check the class names

@Bean Methods

- If @Configuration classes can scan for annotations, just like XML, why use @Bean?
- @Bean methods can have an arbitrary number of parameters (no @Autowired needed)
  - Spring matches them just like constructor parameters
  - Use for advanced configuration that is not easily done declaratively

```
@Configuration
public class ApplicationTestConfig {
    @Bean
    public ImportantService importantService(StringProvider sp) {
        // have opportunity to interact with StringProvider here
        return new ImportantService(sp);
    }
…
```

A suitable bean will automatically be injected here

- Use @Bean when you do not own the source code for the class being instantiated
  - You cannot add annotations to the class's source file

@Bean Methods for DataSource Configuration

- Java configuration is often used in initialize a DataSource

```
@Configuration
@PropertySource("classpath:db.properties")
public class SpringJdbcConfiguration {
   @Autowired
   private Environment env;

   @Bean
   public DataSource createDataSource() {
      DriverManagerDataSource dataSource =
          new DriverManagerDataSource();

      dataSource.setDriverClassName(env.getProperty("db.driver"));
      dataSource.setUrl(env.getProperty("db.url"));
      dataSource.setUsername(env.getProperty("db.username"));
      dataSource.setPassword(env.getProperty("db.password"));
      return dataSource;
   }
}
```

```
@Component
public class DepartmentDao {
     @Autowired
     private DataSource dataSource;
     …
```

Spring calls the @Bean method createDataSource() to satisfy the DataSource dependency

Spring Profiles

- During its development lifecycle, your application will run in different environments
  - Your development environment, the QA/QC environment, the production environment
- Often, beans and property files need to change based on the current environment
  - Need configuration for data sources, different URLs for web services, etc.
- Spring defines *profiles* to support configuration for different environments

```
@Component
@Profile("dev")
public class DevelopmentDbConfig { … }
```

Bean created only if active profile is `dev`

```
@Component
@Profile("prod")
public class ProductionDbConfig { … }
```

Bean created only if active profile is `prod`

- Any bean that does not specify a profile belongs to the `default` profile
- Set the active profile with a Java system property

```
java -Dspring.profiles.active=dev …
```

Set active profile to `dev`

Property Files for Different Environments

- You can load property files customized for different environments with `@PropertySource`
  - Add profile name to file names: `app.`**`prod`**`.properties`, `app.`**`dev`**`.properties`
  - Specify the property file path using an SpEL expression with a default value
    - Syntax: `${`*`expression`*`:`*`default-value`*`}`
    - If *`expression`* is not `null` or `0`, use it; otherwise, use *`default-value`*

```
@Configuration
@PropertySource("classpath:app.${spring.profiles.active:prod}.properties")
public class AppConfig { }
```

If `spring.profile.active` is not set, value is “`prod`”

- Set the active profile for test classes using `@ActiveProfiles`

```
@ActiveProfiles("dev")
public class LibraryDaoTest { … }
```

Chapter Concepts

Managing the Bean Lifecycle

Spring Expression Language (SpEL)

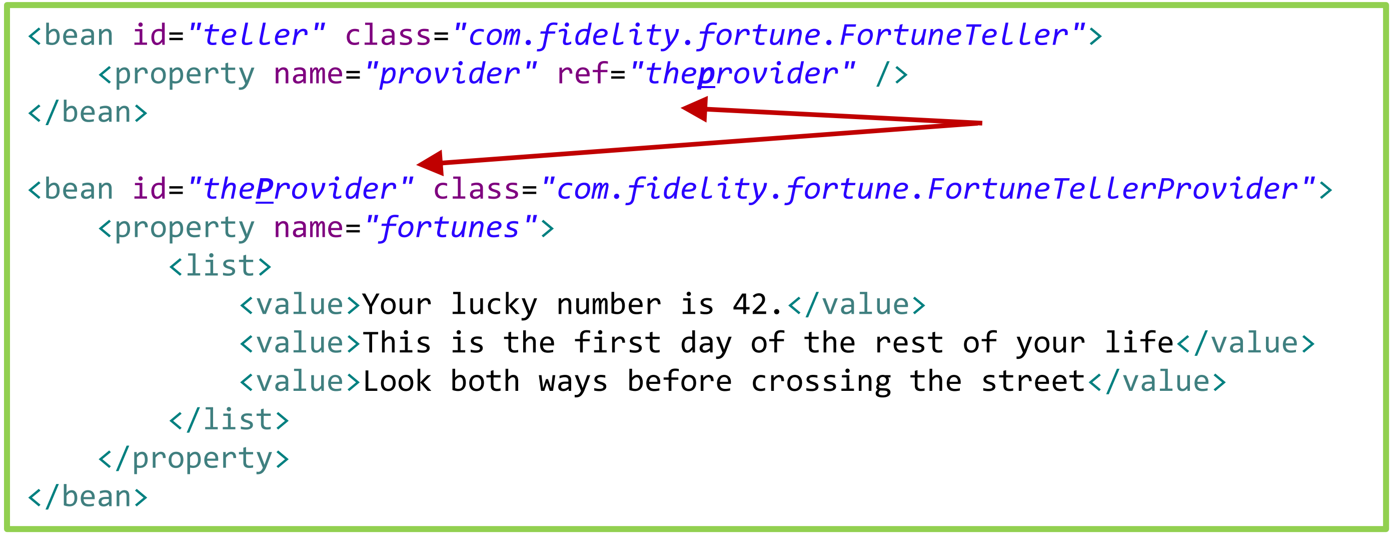
More Configuration Options

Debugging Spring Configuration Problems

Chapter Summary

The Source of the Problem

- The source of many Spring problems is the configuration file (or annotations)
- Misconfigurations, including misnamed beans, are often the culprit
  – Remember the values of the bean `id` and `ref` attributes are case-sensitive



```
<bean id="teller" class="com.fidelity.fortune.FortuneTeller">
    <property name="provider" ref="theprovider" />
</bean>

<bean id="theProvider" class="com.fidelity.fortune.FortuneTellerProvider">
    <property name="fortunes">
        <list>
            <value>Your lucky number is 42.</value>
            <value>This is the first day of the rest of your life</value>
            <value>Look both ways before crossing the street</value>
        </list>
    </property>
</bean>
```

Using a Logging Framework to Debug

- Sometimes, it is necessary to see what Spring is doing when it is attempting to create the beans defined by your configuration settings
  - Use a logging package
    - Set the rootLogger to `DEBUG` level
    - You will see an astounding level of detail

log4j2.properties

```
status = warn
dest = err
name = PropertiesConfig

appender.console.type = Console
appender.console.name = Console
appender.console.target = SYSTEM_OUT
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = Props: %d{HH:mm:ss.SSS} [%t] %-5p %c{36} - %m%n

rootLogger.level = debug
rootLogger.appenderRef.stdout.ref = Console
```

Exercise 3.2: Debugging Spring Configuration Problems

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Concepts

Managing the Bean Lifecycle

Spring Expression Language (SpEL)

More Configuration Options

Debugging Spring Configuration Problems

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The bean lifecycle
- How dependencies are injected
- Dynamically evaluating expressions with Spring Expression Language (SpEL)
- Managing Spring configuration across multiple files
- Debugging Spring configuration problems

Key Points

- The Spring Expression Language allows more complex configuration
- Multiple configuration sources can be used together
- Debugging Spring configuration problems
  - Read the **entire** error message in the stack trace
  - Identify the source of the problem from the stack trace
  - Use a logging framework for those difficult problems

Fidelity LEAP

Technology Immersion Program

Mastering Spring and MyBatis

Chapter 4:
Introduction to MyBatis and Spring

Chapter Overview

In this chapter, we will explore:

- The Domain Store design pattern
- Persisting Java beans with MyBatis
- Configuring and invoking MyBatis from Spring
- Managing relationships in Java and MyBatis

Chapter Concepts

Configuring a DataSource

Domain Store Design Pattern

Configuring MyBatis with Spring

Querying a Database with MyBatis in Spring

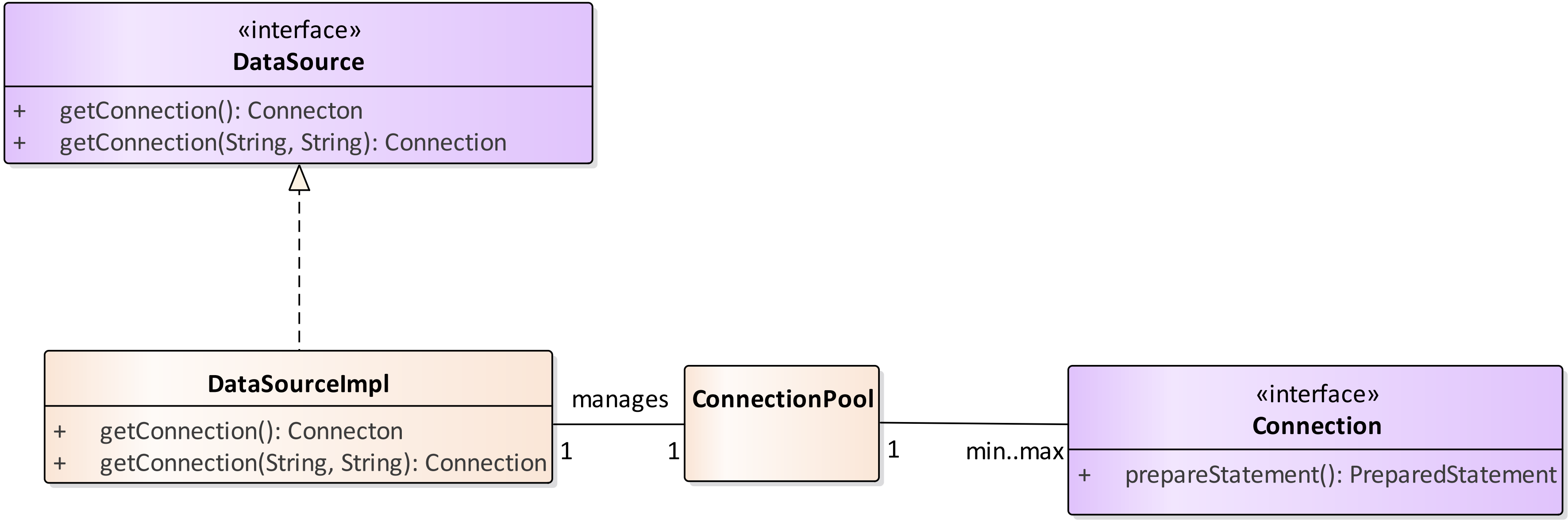
Working with Relationships

Chapter Summary

DataSource

- DataSource is a Java interface
- Spring will create a DataSource instance based on the database properties
- The DataSource will create a pool of database connections
  - The Connections are connected to the database when the pool is created
  - The Connection will be returned to the pool when the `close()` method is called
- We will use this instead of getting a Connection directly from the `DriverManager`
  - Although we will use the aptly named `DriverManagerDataSource`
  - We could replace this with another DataSource implementation
- The connection pool provides a win-win situation:
  1. The overhead of opening and closing database connections is removed since the connections are established with the database when the pool is created
  2. The connections can be shared by multiple clients. When the client code has completed a database operation, it can return the connection to the pool. After the connection has been returned to the pool, another client can obtain it from the DataSource.

DataSource with Connection Pool


«interface»
DataSource
+ getConnection(): Connecton
+ getConnection(String, String): Connection
DataSourceImpl
+ getConnection(): Connecton
+ getConnection(String, String): Connection
manages
1
1
ConnectionPool
1
min..max
«interface»
Connection
+ prepareStatement(): PreparedStatement

Configuring a DataSource with Spring

- Spring simplifies the creation of a `DataSource`

Java Configuration

```java
@Configuration
@PropertySource("classpath:db.properties")
public class SpringJdbcConfiguration {

  @Bean
  public DataSource createDataSource(
                                Environment env) {

      DriverManagerDataSource dataSource =
          new DriverManagerDataSource();

      dataSource.setDriverClassName(env.getProperty("db.driver"));
      dataSource.setUrl(env.getProperty("db.url"));
      dataSource.setUsername(env.getProperty("db.username"));
      dataSource.setPassword(env.getProperty("db.password"));
      return dataSource;
  }
}
```

Spring autowires `@Bean` method parameters

XML Configuration

```xml
<context:property-placeholder location="classpath:db.properties" />

<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${db.driver}" />
    <property name="url" value="${db.url}" />
    <property name="username" value="${db.username}" />
    <property name="password" value="${db.password}" />
</bean>
```

`db.properties`

```
db.url=jdbc:oracle:thin:@localhost:1521:xepdb1
db.driver=oracle.jdbc.driver.OracleDriver
db.username=scott
db.password=TIGER
```

Chapter Concepts

Configuring a DataSource

Domain Store Design Pattern

Configuring MyBatis with Spring

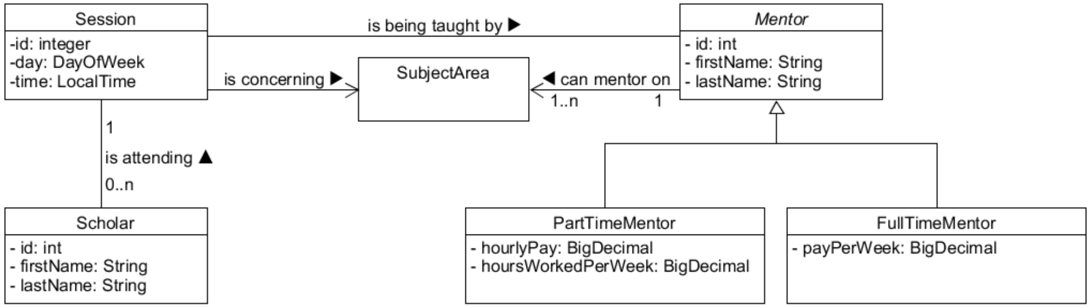
Querying a Database with MyBatis in Spring

Working with Relationships

Chapter Summary

Limitations of JDBC-Based DAOs

- Even a simple object-oriented design can be difficult to translate to a relational model
  - Example: an application that tracks mentoring sessions

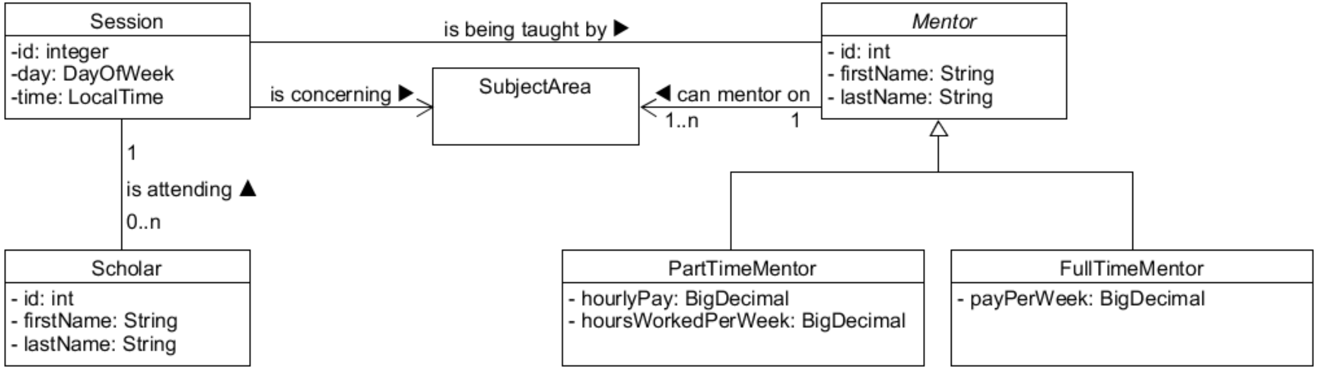


  - Each Session is attended by a group of Scholars
  - Each Session is taught by a single Mentor (ignore the subclasses for now)
  - Each Session concerns a single Subject Area
  - Must retrieve all of these to retrieve a Session

```
public class Session {
    private int id;
    private List<Scholar> scholars;
    private Mentor mentor;
    private DayOfWeek day;
    private LocalTime time;
    private SubjectArea subjectArea;
    …
```

A Complex Object Graph

- Now look at Mentor (still ignoring the subclasses)



Session
-id: integer
-day: DayOfWeek
-time: LocalTime

is being taught by ▶

Mentor
- id: int
- firstName: String
- lastName: String

is concerning ▶

SubjectArea

◀ can mentor on
1..n
1

1
is attending ▲
0..n

Scholar
- id: int
- firstName: String
- lastName: String

PartTimeMentor
- hourlyPay: BigDecimal
- hoursWorkedPerWeek: BigDecimal

FullTimeMentor
- payPerWeek: BigDecimal

  - Each Mentor covers a collection of Subject Areas
  - So, Mentor is also a complex type

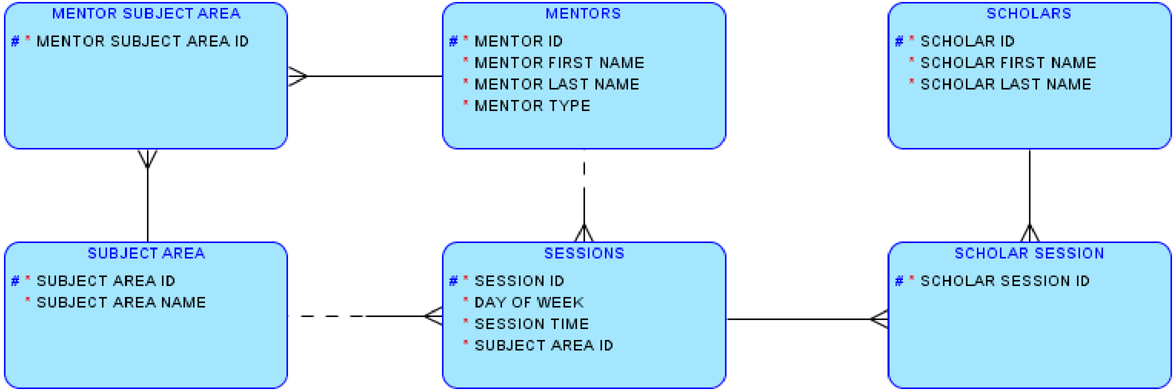
```
public abstract class Mentor {
    private int id;
    private String firstName;
    private String lastName;
    private Set<SubjectArea> subjectAreas;
    …
```

- We call this collection of objects an “object graph”

- How can we represent these in a database?

A Complex Object Graph (continued)

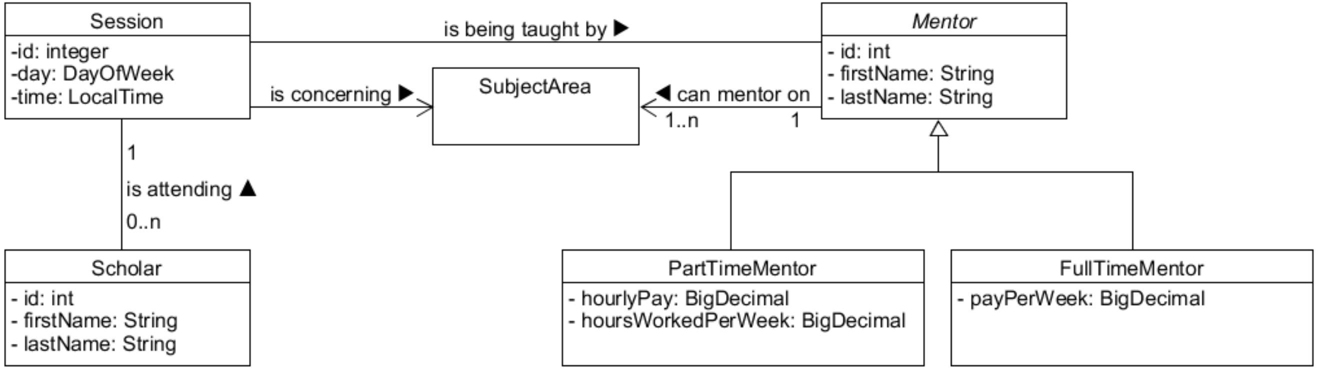
- Here is one possible representation
  - Note the tables resolving many-to-many relationships between:
    - Mentors and Subject Area
    - Sessions and Scholars



- Accessing a Session means reconstructing the object graph from these tables

Inheritance in Relational Databases

- Look at Mentor again, think about the subclasses



- Each Mentor is one of PartTimeMentor or FullTimeMentor
  - There are common fields in Mentor
  - Specialized fields in PartTimeMentor and FullTimeMentor

- How can we represent this in a Relational Database where there is no inheritance?

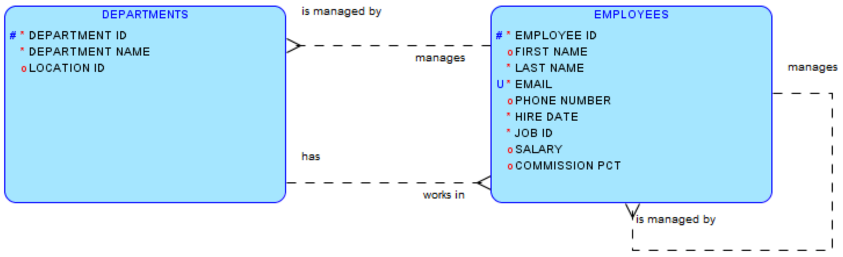
Three Common Approaches to Mapping Inheritance

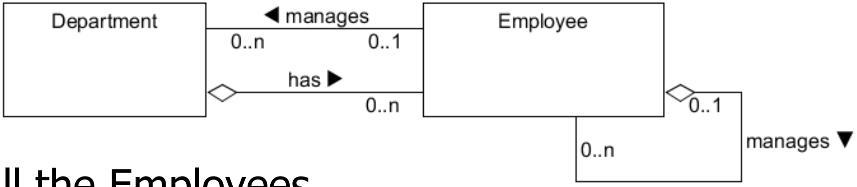
- A single table representing all instances of the superclass (Single Table Inheritance*)
  - One table: Mentor
  - A type column, known as a discriminator, identifies what is held in each row
  - The table contains the superset of all columns from all subclasses (many optional)
- Separate tables for each concrete subclass (Concrete Table Inheritance*)
  - Two tables: FullTimeMentor and PartTimeMentor
  - Common fields are repeated as columns on both tables
  - Any operation in the database that operates on all Mentors requires a `UNION ALL`
  - Unique key management is difficult across the tables
- Separate tables for each class (Class Table Inheritance*)
  - Three tables: Mentor, FullTimeMentor, and PartTimeMentor (our choice in this case)
  - Each table matches the class, primary key is common
  - Retrieving any meaningful data requires joins between these tables
  - Discriminator not required, but may be useful in many cases

*These names are from Fowler, *Patterns of Enterprise Application Architecture*

Recursive Relationships

- Consider the relationship between departments and employees in the HR schema
  - A department may be managed by an employee
  - Many employees may work for a department





- What happens if you retrieve an Employee?
  - Remember that objects should always be fully populated
- Need to retrieve the Employee's department
  - To retrieve a Department, need to retrieve all the Employees
    - Including the one we started with
    - And each Employee has a reference to the Department

Partial Objects

- An object with only some fields loaded is known as a Partial Object
  - Every method must check which fields are loaded
  - Cannot use a simple null check if NULL is a valid value
  - Difficult to prevent this knowledge leaking outside the object

- Object-oriented best practice is always to have fully-populated objects
  - Avoid Partial Objects

- In recursive relationships, it is very expensive to load the whole object graph
  - Need a way to work safely with Partial Objects
  - Solution is a Proxy that intercepts all method calls and ensures data is present if needed

Need for a Persistence Framework

- The previous slides illustrate some effects of the Object-Relational Impedance Mismatch
  - Named for impedance matching in Electrical Engineering
- For very large or highly data-driven applications
  - DAO code can become extremely large and complex when using JDBC
  - Difficult to maintain
  - Difficult to extend
- For such projects, may need way to deal with entire object graphs easily
  - Perhaps load parts of graph only as needed (“lazy loading”)
  - Avoid reloading data when database changes rarely (“caching”)
- Best practice: use a *persistence framework* to persist complex object graphs
  - Instead of writing custom DAO code
  - The Domain Store design pattern

Chapter Concepts

Configuring a DataSource

Domain Store Design Pattern

Configuring MyBatis with Spring

Querying a Database with MyBatis in Spring

Working with Relationships

Chapter Summary

Introducing MyBatis

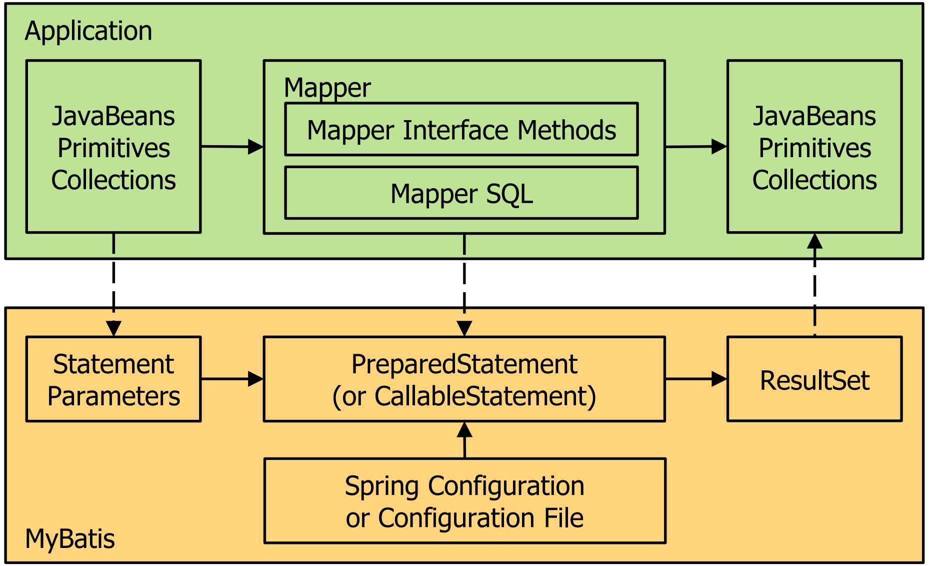
- MyBatis: an open-source SQL mapping framework
  - Makes it easier to create Java objects from database queries
  - Supports relationships between objects: one-to-one, one-to-many, many-to-many
  - Implements advanced features such as lazy loading and caching of query results
- Working with MyBatis:
  1. Write a SQL SELECT statement
  2. Configure MyBatis to map the result set’s columns to Java object properties
  3. Tell MyBatis to execute the query
  4. MyBatis calls constructors and setter methods to create a graph of Java objects
- MyBatis is not a full-fledged Object-Relational Mapping (ORM) framework
  - Doesn’t generate SQL queries for you
  - Gives only limited help with inserting and updating objects in database tables
- But many projects don’t need a heavyweight ORM
  - Teams that hand-tune SQL queries for optimal performance
  - Projects that use stored procedures for most persistence tasks

How MyBatis Works

- Map JavaBeans objects to `PreparedStatement` parameters and `ResultSets`

- The Application view is:
  - Use standard Java types as parameters
  - Execute Java interface methods to access data
  - Receive results as standard Java types

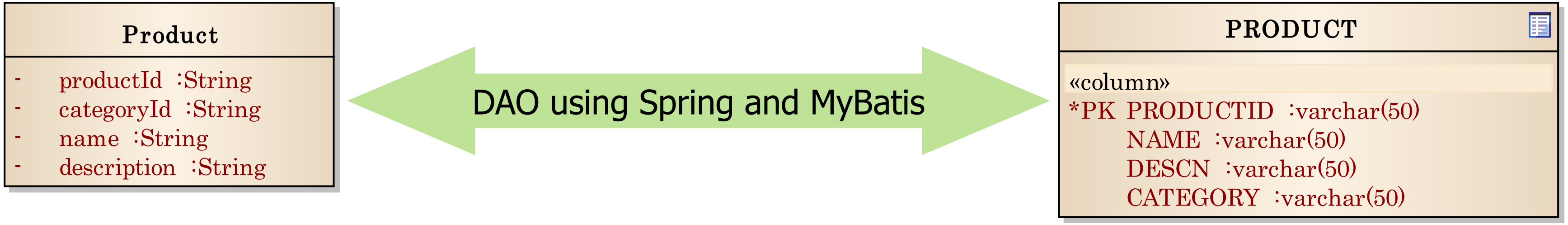
- MyBatis framework will:
  - Create a `PreparedStatement` based on configuration
  - Set parameters on the statement
  - Execute query or update using JDBC
  - Convert `ResultSet` into Java types (usually objects or a collection of objects)



Application
JavaBeans Primitives Collections
Mapper
Mapper Interface Methods
Mapper SQL
JavaBeans Primitives Collections
Statement Parameters
PreparedStatement (or CallableStatement)
ResultSet
Spring Configuration or Configuration File
MyBatis

Steps to Persist an Object with MyBatis and Spring

- Let’s start with a simple example:
  - How to create, read, update, delete (CRUD) Product objects
  - This section shows best practices (not all the possible ways)
    - Will examine the actual requirements later in this chapter



Product

- productId :String
- categoryId :String
- name :String
- description :String

DAO using Spring and MyBatis

PRODUCT

«column»
*PK PRODUCTID :varchar(50)
NAME :varchar(50)
DESCN :varchar(50)
CATEGORY :varchar(50)

- Steps:

1. Configure MyBatis to load mapping files and use a DataSource
2. Write a Java bean to be persisted
3. Create a Java mapping interface that declares the database operations
4. Write an XML mapping file that contains SQL statements
5. Use MyBatis from Service or DAO

Step 1: Configure MyBatis

- You can configure MyBatis with either XML or Java configuration
  - Specify the Java package for MyBatis to scan for mapper interfaces
  - MyBatis will automatically generate objects that implement your mapper interface
  - Spring will then autowire the mapper objects into your Service beans or DAO beans

XML Configuration

```xml
<!-- enable scanning for Spring components and autowiring
    (beware that this does not enable scanning for MyBatis mapper interfaces!) -->
<context:component-scan base-package="com.fidelity.service, com.fidelity.integration" />

<!-- Tell MyBatis where to scan for mappers -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.fidelity.integration" />
</bean>
```

Package of Java interface that maps to MyBatis operations

Java Configuration

```java
@Bean
public MapperScannerConfigurer mapperScannerConfigurer() {
    MapperScannerConfigurer configurer = new MapperScannerConfigurer();
    configurer.setBasePackage("com.fidelity.integration");
    return configurer;
}
```

Step 1: Configure MyBatis (continued)

- Wire up the MyBatis `SqlSessionFactory` in Spring's configuration file
  - `SqlSession` is the core of MyBatis, though we will rarely access directly
  - `configLocation` – location of the MyBatis config file, if required
  - `dataSource` – data source to use
  - `mapperLocations` – locations of MyBatis XML mapping files
  - `typeAliasesPackage` – default Java package name for classes referenced in MyBatis XML mapping file

beans.xml

```
<!-- define the SqlSessionFactory -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mapperLocations" value="classpath*:com/fidelity/**/*.xml" />
    <property name="configLocation" value="classpath*:mybatis-config.xml" />
    <property name="typeAliasesPackage" value="com.fidelity.domain" />
</bean>
```

Often can omit `configLocation` when using MyBatis with Spring

Step 1: Configure MyBatis (Java Configuration)

- You can also configure `SqlSessionFactory` with Java configuration

```
@Configuration
public class MyBatisConfig {
   @Bean
   public SqlSessionFactory sqlSessionFactory(DataSource dataSource,
                                 ResourceLoader resourceLoader) throws Exception {
       Resource[] mapperFiles =
           ResourcePatternUtils.getResourcePatternResolver(resourceLoader)
                               .getResources("classpath*:com/fidelity/**/*.xml");

       SqlSessionFactoryBean factoryBean = new SqlSessionFactoryBean();

       factoryBean.setDataSource(dataSource);
       factoryBean.setMapperLocations(mapperFiles);
       factoryBean.setConfigLocation(new ClassPathResource("mybatis-config.xml"));
       factoryBean.setTypeAliasesPackage("com.fidelity.domain");

       return factoryBean.getObject();
}
```

Spring automatically injects these parameters

Set the session factory's properties

Return the session factory

MyBatis Configuration File

- By default, the MyBatis configuration file is named `mybatis-config.xml`
- When using MyBatis standalone, this defines parameters for `SqlSessionFactoryBean`:
  - Data source
  - Mapper locations
  - Type aliases
- Usually not needed when using MyBatis with Spring, but can be used:
  - To set mapper locations and type aliases in addition to those in the bean declaration
  - To set global parameters such as caching, lazy loading, automapping, timeouts

Spring Support for MyBatis

- Spring integrates nicely with MyBatis
  - MyBatis provides template classes that integrate with Spring
    - Simplifies getting access to mappers
  - Handles `SQLException` and maps exceptions

- Spring simplifies the configuration of MyBatis
  - E.g., configure data source from Spring configuration file

- Spring also provides integrated transaction management
  - Can add transactions to methods using Spring's Aspect-Oriented Programming (AOP)
  - Simply add Spring's `@Transactional` to a class or method
    - Application code does not explicitly begin, commit, or roll back transactions
    - It is still happening behind the scenes controlled by Spring

Exercise 4.1: Configure MyBatis with Spring

HANDS-ON EXERCISE

15 min

- Follow the directions in your Exercise Manual

Chapter Concepts

Configuring a DataSource

Domain Store Design Pattern

Configuring MyBatis with Spring

Querying a Database with MyBatis in Spring

Working with Relationships

Chapter Summary

Step 2: Java Object to Persist

1. Configure MyBatis to load mapping files and use a DataSource
2. **Write a Java bean to be persisted**
3. Create a Java mapping interface that declares the database operations
4. Write an XML mapping file that contains SQL statements
5. Use MyBatis from Service or DAO

```
public class Product {

  private int productId;
  private int categoryId;
  private String name;
  private String description;

  // getters and setters
}
```

MyBatis will use the default constructor and setter methods, if present

If no zero-argument constructor, MyBatis will inject arguments for another constructor

If no setter for a field, MyBatis will access the field directly (even if private)

Step 3: MyBatis Mapping Interface

1. Configure MyBatis to load mapping files and use a DataSource
2. Write a Java bean to be persisted
3. **Create a Java mapping interface that declares the database operations**
4. Write an XML mapping file that contains SQL statements
5. Use MyBatis from Service or DAO

ProductMapper.java

```java
public interface ProductMapper {
    Product getProduct(int productId);
    List<Product> getProductListByCategory(int categoryId);
    void insertProduct(Product product);
}
```

- Create a Java interface
  - Methods declare the database operations for the Java object that will be persisted
- Instances of this interface will be created by `SqlSession`
  - You do not have to write a class that implements this interface!!!

Step 4: MyBatis XML Mapping File

1. Configure MyBatis to load mapping files and use a DataSource
2. Write a Java bean to be persisted
3. Create a Java mapping interface that declares the database operations
4. **Write an XML mapping file that contains SQL statements**
5. Use MyBatis from Service or DAO

- Mapping file describes how database tables and columns are mapped to classes and fields

- Some important things to remember:
  - The mapping file must be consistent with the mapping interface
  - The database operations defined in the mapping file must correspond to the methods in the interface
  - The operation ids must match the method names

Step 4: MyBatis Mapping File Example

- By convention, the name of mapping file is *ClassName***Mapper**.xml

ProductMapper.xml

```java
public class Product {

  public int getProductId() { … }
  public String getName() { … }
  public int getCategoryId() { … }
  public String getDescription() { … }
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
                        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
  <mapper namespace="com.fidelity.integration.ProductMapper">

    <select id="getProduct" resultType="Product">
        SELECT productId, name, descn as description, category as categoryId
        FROM   product
        WHERE  productid = #{productId}
    </select>

    <select id="getProductListByCategory" resultType="Product">
        SELECT productId, name, descn as description, category as categoryId
        FROM   product
        WHERE  category = #{value}
    </select>

    <insert id="insertProduct" parameterType="Product">
        INSERT INTO product (productid, name, descn, category)
        VALUES (#{productId}, #{name}, #{description}, #{categoryId})
    </insert>
</mapper>
```

`<select>` id must match method name in mapper interface

If column name doesn't match property name exactly, define an alias

Step 4: Mapping File Location

- Place the XML mapping file in the location defined in the Spring configuration file
  - Usually relative to classpath of application
    - With package qualified name to avoid conflicts
  - When using Maven, put it in a subdirectory of `src/main/resources`

- Could place it in the DAO's package
  - If DAO is in `com.fidelity.integration`, mapping file will be at:

```
classpath*:/com/fidelity/integration/ProductMapper.xml
```

  - Another common option is to place it in the package of business object

```
classpath*:/com/fidelity/domain/ProductMapper.xml
```

- By using ** in the configuration file, specify any number of intermediate directories

```
classpath*:com/fidelity/**/*.xml
```

Step 5: Use MyBatis in Service or DAO

1. Configure MyBatis to load mapping files and use a DataSource
2. Write a Java bean to be persisted
3. Create a Java mapping interface that declares the database operations
4. Write an XML mapping file that contains SQL statements
5. **Use MyBatis from Service or DAO**

- The DAO class calls on the Mapper class
  - To interact with the database
- Notice that the DAO class has Spring auto-wire the Mapper bean

`@Primary` is required if mapper interface is in same package as DAO

```
@Primary
@Repository("productDao")
public class ProductDaoMyBatisImpl implements ProductDao {
   @Autowired
   private ProductMapper mapper;

   @Override
   public List<Product> getProducts(int categoryId) {
      return mapper.getProductListByCategory(categoryId);
   }
}
```

Easy to see why we often don't need the DAO when using MyBatis: treat the mapper as the DAO instead

Exercise 4.2:
Query the Database with MyBatis and Spring

HANDS-ON EXERCISE

30 min

- Follow the directions in your Exercise Manual

Chapter Concepts

Configuring a DataSource

Domain Store Design Pattern

Configuring MyBatis with Spring

Querying a Database with MyBatis in Spring

Working with Relationships

Chapter Summary

ResultMaps

- Best practice: use a `ResultMap` to define the mapping instead of column aliases
  – `ResultMaps` allow more complex mappings to be defined

```
<mapper namespace="com.fidelity.integration.ProductMapper">
    <resultMap type="Product" id="ProductMap">
        <id     property="productId"   column="PRODUCTID"/>
        <result property="name"        column="NAME"/>
        <result property="description" column="DESCN"/>
        <result property="categoryId"  column="CATEGORY"/>
    </resultMap>

    <select id="getProducts" resultType="Product">
        SELECT productid, name, descn as description, category as categoryId
        FROM   product
    </select>

    <select id="getProduct" resultMap="ProductMap">
        SELECT productid, name, descn, category
        FROM   product
        WHERE  productid = #{productId}
    </select>
```

`<id>` identifies table's primary key

Column names are not case-sensitive

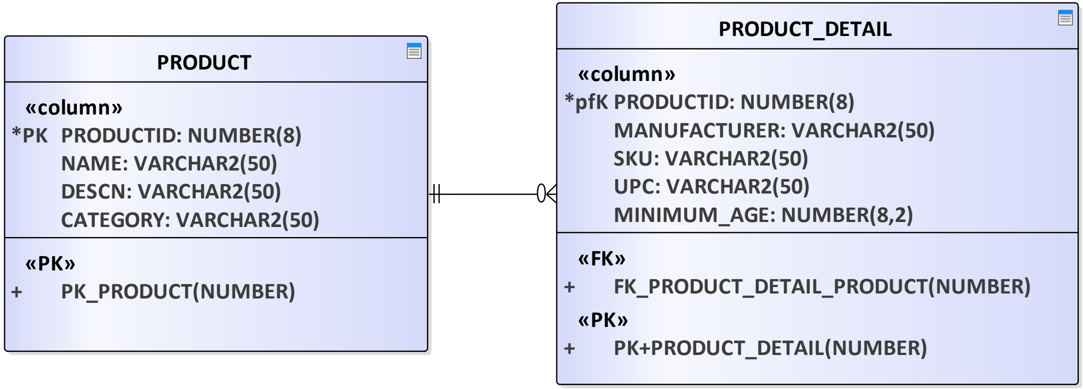
Property names are case-sensitive

Without `ResultMap`, must define column aliases

With `ResultMap`, use raw column names

One-to-One Mapping (DB)

- Database:
  - PRODUCT_DETAIL table has PRODUCTID column that is foreign key to PRODUCT and is also primary key
- Java:
  - Product class has reference to a ProductDetail object



PRODUCT

«column»
*PK PRODUCTID: NUMBER(8)
NAME: VARCHAR2(50)
DESCN: VARCHAR2(50)
CATEGORY: VARCHAR2(50)

«PK»
+ PK_PRODUCT(NUMBER)

PRODUCT_DETAIL

«column»
*pfK PRODUCTID: NUMBER(8)
MANUFACTURER: VARCHAR2(50)
SKU: VARCHAR2(50)
UPC: VARCHAR2(50)
MINIMUM_AGE: NUMBER(8,2)

«FK»
+ FK_PRODUCT_DETAIL_PRODUCT(NUMBER)

«PK»
+ PK+PRODUCT_DETAIL(NUMBER)

```
public class Product {
    private int productId;
    private ProductDetail detail;
    …
}
```

This field defines the one-to-one relationship

```
public class ProductDetail {
    private int productId;
    private int manufacturer;
    private String sku;
    …
}
```

One-to-One Mapping (MyBatis)

- Performing the `getProductsWithDetail` returns fully populated `Product`s
- This technique works, but probably is not what we would use in production
  - If we already have `Product` mappings defined, we want to reuse them

```
<resultMap type="Product" id="ProductWithDetailMap">
    <id     property="productId"           column="PRODUCTID"/>
    <result property="name"                column="NAME"/>
    <result property="description"         column="DESCN"/>
    <result property="categoryId"          column="CATEGORY"/>
    <result property="detail.productId"    column="PRODUCTID"/>
    <result property="detail.manufacturer" column="MANUFACTURER"/>
    <result property="detail.sku"          column="SKU"/>
    <result property="detail.upc"          column="UPC"/>
    <result property="detail.minimumAge"   column="MINIMUM_AGE"/>
</resultMap>

<select id="getProductsWithDetail" resultMap="ProductWithDetailMap">
    SELECT  p.productid, p.name, p.descn, p.category, d.manufacturer, d.sku, d.upc, d.minimum_age
      FROM  product p
      JOIN  product_detail d ON p.productid = d.productid
</select>
```

One-to-One Mapping by Extension

- One map can extend another

```
<resultMap type="Product" id="ProductWithDetailMap">
    <id     property="productId"             column="PRODUCTID"/>
    <result property="name"                  column="NAME"/>
    <result property="description"           column="DESCN"/>
    <result property="categoryId"            column="CATEGORY"/>
    <result property="detail.productId"      column="PRODUCTID"/>
    <result property="detail.manufacturer"   column="MANUFACTURER"/>
    <result property="detail.sku"            column="SKU"/>
    <result property="detail.upc"            column="UPC"/>
    <result property="detail.minimumAge"     column="MINIMUM_AGE"/>
</resultMap>
```

```
<resultMap type="Product" id="ProductMap">
    <id     property="productId"   column="PRO
    <result property="name"        column="NAM
    <result property="description" column="DES
    <result property="categoryId"  column="CAT
</resultMap>

<resultMap type="Product" id="ProductWithDetailByExtension" extends="ProductMap">
    <result property="detail.productId"    column="PRODUCTID"/>
    <result property="detail.manufacturer" column="MANUFACTURER"/>
    <result property="detail.sku"          column="SKU"/>
    <result property="detail.upc"          column="UPC"/>
    <result property="detail.minimumAge"   column="MINIMUM_AGE"/>
</resultMap>
```

- But if we need the detail table in other places, we still must repeat the mappings
  - – We can avoid this by using a nested `ResultMap` or a nested SELECT

One-to-One Mapping with Nested ResultMaps

- Define a `ResultMap` for a `Product` with an `<association>` element
  - The association data will be loaded using a single query

```
<resultMap type="Product" id="ProductWithNestedDetailMap">
    <id     property="productId"    column="PRODUCTID"/>
    <result property="name"         column="NAME"/>
    <result property="description"  column="DESCN"/>
    <result property="categoryId"   column="CATEGORY"/>
    <association property="detail"  resultMap="ProductDetailMap" />
</resultMap>

<resultMap type="ProductDetail" id="ProductDetailMap">
    <id     property="productId"    column="PRODUCTID"/>
    <result property="manufacturer" column="MANUFACTURER"/>
    <result property="sku"          column="SKU"/>
    <result property="upc"          column="UPC"/>
    <result property="minimumAge"   column="MINIMUM_AGE"/>
</resultMap>

<select id="getProductsWithNestedDetail" resultMap="ProductWithNestedDetailMap">
    SELECT  p.productid, p.name, p.descn, p.category, d.manufacturer,
            d.sku, d.upc, d.minimum_age
      FROM  product p
      JOIN  product_detail d ON p.productid = d.productid
</select>
```

`<association>` means "one-to-one"

Reference to another `<resultMap>`

Combine Extension and Nesting

- Can combine both techniques to maximize re-use
  - This is the best practice for a production solution

```
<resultMap type="Product" id="ProductWithNestedDetailMap">
    <id     property="productId"    column="PRODUCTID"/>
    <result property="name"         column="NAME"/>
    <result property="description"  column="DESCN"/>
    <result property="categoryId"   column="CATEGORY"/>
    <association property="detail" resultMap="ProductDetailMap" />
</resultMap>
```

```
<resultMap type="Product" id="ProductMap">
    <id     property="productId"   column="PRODUCTID"/>
    <result property="name"        column="NAME"/>
    <result property="description" column="DESCN"/>
    <result property="categoryId"  column="CATEGORY"/>
</resultMap>

<resultMap type="Product" id="ProductWithNestedDetailByExtension" extends="ProductMap">
    <association property="detail" resultMap="ProductDetailMap" />
</resultMap>
```

`Product` map without the one-to-one relationship

`Product` map with the one-to-one relationship

One-to-One Mapping with Nested Select

- Can define an `<association>` to use another `<select>` statement instead of a JOIN
  - Set the `<association>`'s `select` attribute to the nested `<select>`
  - A separate SELECT will be executed to fetch the associated `ProductDetail`

```
<resultMap type="Product" id="ProductWithNestedDetailSelect">
    <id     property="productId"    column="PRODUCTID"/>
    <result property="name"         column="NAME"/>
    <result property="description"  column="DESCN"/>
    <result property="categoryId"   column="CATEGORY"/>
    <association property="detail"  column="PRODUCTID" select="getProductDetail" />
</resultMap>

<select id="getProductsWithNestedSelect" resultMap="ProductWithNestedDetailSelect">
    SELECT productid, name, descn, category
    FROM   product
</select>

<select id="getProductDetail" parameterType="int" resultMap="ProductDetailMap">
    SELECT productid, manufacturer, sku, upc, minimum_age
    FROM   product_detail
    WHERE  productid = #{value}
</select>
```

One-to-Many Mapping with Nested ResultMap

- The relationship between Category and Product is a one-to-many relationship
  - Each Product has a Category
  - Each Category may have many Products
- The `<collection>` element can be used to define this in MyBatis

This field defines the one-to-many relationship

```
<resultMap type="Category" id="CategoryWithNestedProductMap">
    <id     property="categoryId"   column="ID"/>
    <result property="name"         column="CAT_NAME"/>
    <collection property="products" resultMap="ProductMap" />
</resultMap>
```

`<collection>` means “one-to-many”

```
public class Category {
   private Set<Product> products;
```

```
public class Product {
   private Category category;
```

```
<select id="getCategoriesWithNestedProduct" resultMap="CategoryWithNestedProductMap">
    SELECT p.productid, p.name, p.descn, p.category, c.id, c.name AS cat_name
    FROM   category c
    LEFT OUTER JOIN
           product p
    ON     p.category = c.id
</select>
```

These two columns have the same value. Since this is an outer join, must put the id in twice (once from `category` and once from `product`), otherwise, since the id is not null from `category`, MyBatis will create `Product`s even when all the other details are null

One-to-Many Mapping with Nested Select

- Can define a nested SELECT instead of a JOIN
  – May hurt performance: a nested SELECT will be executed for each child row

```
<resultMap type="Category" id="CategoryWithNestedProductSelect">
    <id      property="categoryId"   column="ID"/>
    <result property="name"         column="CAT_NAME"/>
    <collection property="products" column="ID" select="getProductListByCategory" />
</resultMap>

<select id="getProductListByCategory" resultType="Product">
    SELECT productid, name, descn as description, category as categoryId
    FROM   product
    WHERE  category = #{value}
</select>

<select id="getCategoriesWithNestedSelect" resultMap="CategoryWithNestedProductSelect">
    SELECT id, name AS cat_name
    FROM   category
</select>
```

Exercise 4.3: Query for Complex Object Relationships

HANDS-ON EXERCISE

45 min

- Follow the directions in your Exercise Manual

Chapter Concepts

Configuring a DataSource

Domain Store Design Pattern

Configuring MyBatis with Spring

Querying a Database with MyBatis in Spring

Working with Relationships

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The Domain Store design pattern
- Persisting Java beans with MyBatis
- Configuring and invoking MyBatis from Spring
- Managing relationships in Java and MyBatis

Key Points

- MyBatis simplifies JDBC code through a mapping file
  - Built around prepared statements
- To use MyBatis + Spring:
  - Configure MyBatis to load mapping files and use a DataSource
  - Write a Java bean
  - Create a mapping file interface that declares the database operations
  - Write mapping file for the Java Bean
  - Use MyBatis from Service or DAO (inject mapper)
- MyBatis supports relationships
  - One-to-one, one-to-many, and many-to-many

Fidelity LEAP

Technology Immersion Program

Mastering Spring and MyBatis

Chapter 5: Working Effectively with MyBatis

Chapter Overview

In this chapter, we will explore:

- Performing database update operations with MyBatis
- Managing transactions in JUnit tests with Spring
- Configuring MyBatis with annotations
- Using MyBatis custom type handlers and query caching
- Using advanced MyBatis features

Chapter Concepts

DML Through MyBatis with XML

Transaction Management in Testing

SQL Mappers Using Annotations

Advanced Topics

Chapter Summary

Mapped Statements

- MyBatis supports DML operations
  - INSERT
  - UPDATE
  - DELETE
- The following slides show examples of each type of statement and how to use them

INSERT

- An SQL INSERT command is configured with an `<insert>` element

Notice use of `Product` property names

```
    <insert id="insertProduct" parameterType="Product">
        INSERT INTO product (productid, name, descn, category)
        VALUES (#{productId}, #{name}, #{description}, #{categoryId})
    </insert>
```

- The INSERT is executed by using the Mapper interface object
  - Mapper method may be defined as returning `int` (number of rows affected) or `void`

```
public interface ProductMapper {
    int insertProduct(Product product);
    …
```

Returns number of rows inserted

```
@Service
public class ProductService {
    @Transactional
    public boolean insertProduct(Product product) {
        return mapper.insertProduct(product) == 1;
    }
```

Autogenerated Primary Keys

- MyBatis can work with primary keys that are automatically generated by the database
  - Available if database supports IDENTITY columns
  - Add useGeneratedKeys, keyProperty, and keyColumn to <insert> statement

```
<insert id="insertProductWithIdentity" parameterType="Product"
        useGeneratedKeys="true" keyProperty="productId" keyColumn="productid">
    INSERT INTO product2 (name, descn, category)
    VALUES (#{name}, #{description}, #{categoryId})
</insert>
```

- MyBatis updates the key value of the object passed in

```
Product product = new Product(0,
         "Jet Pack", "Personal flight device", 65);

dao.insertProductWithIdentity(product);

int generatedId = product.getProductId();
```

Initialize the object with a dummy product id value

Product id is now the key value generated by the database

Primary Keys from Sequences

- Older versions of Oracle do not support `IDENTITY` columns
  - Instead, keys were generated by *sequences*
- If the sequence is used in a trigger, `useGeneratedKeys` works
  - Otherwise, add a `<selectKey>` element to the `<insert>` statement

```
<insert id="insertProductWithSequence" parameterType="Product">
    <selectKey keyProperty="productId" resultType="int" order="BEFORE">
        SELECT product2_seq.NEXTVAL FROM DUAL
    </selectKey>
    INSERT INTO product2 (productid, name, descn, category)
    VALUES (#{productId}, #{name}, #{description}, #{categoryId})
</insert>
```

- As before, MyBatis updates the key value of the object passed in

UPDATE

- An SQL UPDATE command is configured with an `<update>` element

```
<update id="updateProduct" parameterType="Product">
    UPDATE product
    SET    name = #{name}, descn = #{description}, category = #{categoryId}
    WHERE  productid = #{productId}
</update>
```

- The UPDATE is executed by using the Mapper interface object

Returns number of rows updated

```
public interface ProductMapper {
    int updateProduct(Product product);
    …
}
```

```
@Service
public class ProductService {
    @Transactional
    public boolean updateProduct(Product product) {
        return mapper.updateProduct(product) == 1;
    }
```

In this case, the update is by primary key, but that may not always be the case, so you may prefer to return the row count

DELETE

- A SQL DELETE command is configured with a `<delete>` element

```
<delete id="deleteProduct" parameterType="int">
    DELETE FROM product
    WHERE productid = #{value}
</delete>
```

- The DELETE is executed by using the Mapper interface object

Returns number of rows deleted

```
public interface ProductMapper {
    int deleteProduct(int productId);
    …
}
```

As before, you may prefer to return the row count

```
@Override
@Transactional
public boolean deleteProduct(int productId) {
    return mapper.deleteProduct(productId) == 1;
}
```

Cascading Inserts

- Unlike an ORM framework, MyBatis does not support *cascading inserts*
  - Cascading insert example: when inserting a department, insert all its employees as well
  - With MyBatis, you need to call multiple methods explicitly to perform all inserts

```java
public class Department {
  public Set<Employee> getEmployees() {…}
```

```java
@Repository
public class EmployeeDao {
  @Autowired
  private EmployeeMapper mapper;

  public void insertDepartment(Department d) { … }
  public void insertEmployee(Employee e) { … }
```

```java
@Service
public class DepartmentService {
   @Autowired
   private EmployeeDao dao;

   @Transactional
   public void addDepartment(Department dept) {

      dao.insertDepartment(dept);

      for (Employee emp : dept.getEmployees()) {
         dao.insertEmployee(emp);
      }
   }
}
```

Service class, not the DAO, defines transactions

Insert the department

Insert each `Employee` individually

Chapter Concepts

DML Through MyBatis with XML

Transaction Management in Testing

SQL Mappers Using Annotations

Advanced Topics

Chapter Summary

Transaction Manager

- Spring simplifies transactions
  - Define a transaction manager to use the DataSource
  - The transaction manager will control starting and committing transactions

```xml
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="url" value="${db.url}" />
    <property name="driverClassName" value="${db.driver}" />
    <property name="username" value="${db.username}" />
    <property name="password" value="${db.password}" />
</bean>

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<!-- enable transaction demarcation with annotations -->
<tx:annotation-driven />
```

Transactions Through Annotations

- Use Annotations to apply transaction boundaries
  - The Spring transaction manager automatically manages the transaction
    - Starts the transaction when the method is called
    - Commits the transaction when the method completes
    - Or rolls the transaction back if an **unchecked** exception is thrown

```
public class ProductBusinessService {

   @Transactional
   public boolean insertProduct(Product product) {
      …  // validate the Product, etc.

      return dao.insertProduct(product);
   }
```

Spring begins a transaction when this method is called

If the method throws a `RuntimeException`, Spring rolls back the transaction; if no exception is thrown, Spring commits the transaction

Testing and Transactions

- Spring provides support for managing transactions in tests with its TestContext framework
  - Remember, in JUnit use `@ExtendWith(SpringExtension.class)`
- You must do the following:
  - Provide a `PlatformTransactionManager` bean
    - For example, in the `beans.xml` file that is loaded by `@ContextConfiguration`
  - Annotate your JUnit test class with the `@Transactional` annotation
- The TestContext causes all `@Transactional` test methods to rollback automatically
  - If necessary, use `@Rollback(false)` or `@Commit` to override this default behavior
- The Spring documentation provides more detail on the options that are available
  - https://docs.spring.io/spring-framework/docs/current/reference/html/testing.html

Transaction Management in Tests

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(locations="classpath:product-beans.xml")
@Transactional
class ProductDaoMyBatisImplTest {
    @Autowired
    private ProductDao dao;

    @Test
    void testGetProducts() {
        …
    }

    @Test
    void testInsertProduct() {
        …
    }

    @Test
    void testAnotherThing() {
        …
    }
}
```

Annotate class or methods with `@Transactional`

These test methods will roll back automatically

Transaction Management in Tests (continued)

- `@BeforeEach` and `@AfterEach` run inside any transaction for transactional tests
- In addition, there are annotations that run outside the transaction for each test
  - `@BeforeTransaction` runs before the transaction starts
  - `@AfterTransaction` runs after the transaction has ended (usually rolled back)

@DirtiesContext

- The TestContext caches contexts
  - Loaded only once per "suite" (Spring interprets this as "JVM instance")
  - All tests run through Maven are run in the same JVM instance and bean factory
  - Singleton beans are not re-initialized when a second test case is executed
- Sometimes, your tests override one of a bean's dependencies
  - For example, to load a mock version of a dependency
  - This action renders the context unreliable for other tests
- Use `@DirtiesContext` to indicate operations that leave the context in an unreliable state
  - The `@DirtiesContext` tells the testing framework to close and recreate the context for later tests
  - Do this as little as possible: contexts are cached for a reason
  - Consider gathering such tests together

Exercise 5.1: DML with MyBatis and Spring

HANDS-ON EXERCISE

60 min

- Follow the directions in your Exercise Manual

Chapter Concepts

DML Through MyBatis with XML

Transaction Management in Testing

SQL Mappers Using Annotations

Advanced Topics

Chapter Summary

Mapped Statements

- MyBatis defines annotations for different statements
  - These are used in the Mapper interface method definition

```
public interface ProductMapper {
  @Select("""
     SELECT  productid,
             name,
             descn as description,
             category as categoryId
     FROM  product
     """)
  List<Product> getProducts();

  @Insert("""
     INSERT INTO product
            (productid, name, descn, category)
     VALUES (#{productId}, #{name},
             #{description}, #{categoryId})
     """)
  int insertProduct(Product product);
```

```
@Update("""
    UPDATE product
       SET name = #{name},
           descn = #{description},
           category = #{categoryId}
     WHERE productid = #{productId}
     """)
int updateProduct(Product product);

@Delete("""
    DELETE FROM product
          WHERE productid = #{value}
    """)
int deleteProduct(int productId);
```

Autogenerated Primary Keys with Annotations

- To use autogenerated primary keys:
  - Use the `@Options` annotation
    - With the `useGeneratedKeys` and `keyProperty` attributes

```
@Insert("INSERT INTO product2 (name, descn, category) " +
        "VALUES (#{name}, #{description}, #{categoryId})")
@Options(useGeneratedKeys=true, keyProperty="productId", keyColumn="productid")
int insertProductWithIdentity(Product product);
```

- To use Oracle sequences for generating primary keys:
  - Use the `@SelectKey` annotation
    - With the `statement`, `resultType`, `before` and `keyProperty` attributes

```
@Insert("INSERT INTO product2 (productid, name, descn, category) " +
        "VALUES (#{productId}, #{name}, #{description}, #{categoryId})")
@SelectKey(statement="SELECT product2_seq.NEXTVAL FROM DUAL", keyProperty="productId",
           resultType=int.class, before=true)
int insertProductWithSequence(Product product);
```

ResultMaps

- You can map query results to JavaBean properties
  – Use the `@Result` annotation

```
@Select("SELECT productid, name, descn, category " +
        "  FROM product " +
        " WHERE productid = #{productId}")
@Result(property="productId",   column="PRODUCTID", id=true)
@Result(property="name",        column="NAME")
@Result(property="description", column="DESCN")
@Result(property="categoryId",  column="CATEGORY")
Product getProduct(int productId);
```

MyBatis substitutes value of method parameter

  – You can also use a `ResultMap` defined in a Mapper XML file

```
@Select("SELECT productid, name, descn, category " +
        "  FROM product " +
        " WHERE category = #{value}")
@ResultMap("com.fidelity.integration.ProductMapper.ProductMap")
List<Product> getProductListByCategory(int categoryId);
```

One-to-One Mapping

- To load a one-to-one association, use the `@One` annotation
  - Uses a nested select statement
  - Nested `ResultMap` (join mapping) is not supported—if needed, use `@ResultMap`

```
@Select("SELECT productid, name, descn, category " +
        "  FROM product")
@Result(property="productId",   column="PRODUCTID", id=true)
@Result(property="name",        column="NAME")
@Result(property="description", column="DESCN")
@Result(property="categoryId",  column="CATEGORY")
@Result(property="detail",      column="PRODUCTID",
        one=@One(select="getProductDetail"))
List<Product> getProductsWithNestedSelect();

@Select("SELECT productid, manufacturer, sku, upc, minimum_age " +
        "  FROM product_detail " +
        " WHERE productid = #{value}")
@Result(property="productId",    column="PRODUCTID", id=true)
@Result(property="manufacturer", column="MANUFACTURER")
@Result(property="sku",          column="SKU")
@Result(property="upc",          column="UPC")
@Result(property="minimumAge",   column="MINIMUM_AGE")
ProductDetail getProductDetail(int productId);
```

Column value to pass as argument to `getProductDetail()`

Name of mapper method that defines nested SELECT

One-to-Many Mapping

- To load a one-to-many association, use the `@Many` annotation
  - Uses a nested SELECT statement
    - Again, nested result (join mapping) is not available through annotations
    - If needed, use `@ResultMap` annotation to reference result map in XML mapping file

```
@Select("SELECT id, name " +
        "  FROM category")
@Result(property="categoryId",  column="ID", id=true)
@Result(property="name",        column="NAME")
@Result(property="products",    column="ID",
        many=@Many(select="getProductListByCategory"))
List<Category> getCategoriesWithNestedSelect();

@Select("SELECT … FROM product " +
        " WHERE category = #{categoryId}")
@Result(…)
…
List<Product> getProductListByCategory(int categoryId);
```

Column value to pass as argument to `getProductListByCategory()`

Exercise 5.2: Using Annotations with MyBatis

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Concepts

DML Through MyBatis with XML

Transaction Management in Testing

SQL Mappers Using Annotations

Advanced Topics

Chapter Summary

Embedded Databases

- Spring provides first-class support for embedded databases
  - HyperSQL, the default, http://hsqldb.org/
  - H2, `type="H2"`, http://www.h2database.com
  - Apache Derby, `type="DERBY"`, https://db.apache.org/derby/
    - A version of Derby ships with the JDK as Java DB
  - Others by extension

pom.xml

```
<!-- HyperSQL In-memory Database -->
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>2.7.1</version>
    <scope>test</scope>
</dependency>
```

```
<beans …
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="…
    http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
     …">

    <!-- Define a DataSource for an in-memory database -->
    <jdbc:embedded-database id="hsqldbDataSource">
        <jdbc:script location="classpath:products-hsqldb-schema.sql" />
        <jdbc:script location="classpath:products-hsqldb-dataload.sql" />
    </jdbc:embedded-database>
```

Scripts are executed in order

Defining Data Sources with Profiles

- Use Spring profiles to define different data sources for different environments
  - Depending on the active profile, only one bean with id `dataSource` will be created

db-beans-prod.xml

```
<beans profile="prod">

   <!-- Define a production DataSource for Oracle -->
   <bean id="dataSource"
         class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="url" value="${db.url}" />
      <property name="driverClassName" value="${db.driver}" />
      <property name="username" value="${db.username}" />
      <property name="password" value="${db.password}" />
   </bean>
</beans>
```

beans.xml

```
<bean id="sqlSessionFactory"
      class="org.mybatis.spring.SqlSessionFactoryBean">
   <property name="dataSource" ref="dataSource" />
```

References the correct `dataSource` for the current environment

db-beans-dev.xml

```
<beans profile="dev">
   <!-- Define a test DataSource for an in-memory database -->
   <jdbc:embedded-database id="dataSource">
      <jdbc:script location="classpath:products-hsqldb-schema.sql" />
      <jdbc:script location="classpath:products-hsqldb-dataload.sql" />
   </jdbc:embedded-database>
</beans>
```

```
BeanFactory beanFactory =
  new ClassPathXmlApplicationContext(
    "beans.xml","db-beans-prod.xml","db-beans-dev.xml");
```

Load all config files

Set the active profile when you run the application

```
$  java -Dspring.profiles.active=dev …
```

Things to Remember When Using Embedded Databases

- Different dialects of SQL
  - HyperSQL is more standards compliant than Oracle, Oracle has more features
  - `NUMERIC` instead of `NUMBER`, `VARCHAR` instead of `VARCHAR2`, `DATE` only holds a date
  - In general, DML and query behavior is more consistent than DDL
- `ORDER BY`
  - Queries only return a defined order if there is an `ORDER BY`
  - Within a given engine, queries ***may*** be consistent from run to run with a given dataset
  - Switching engine will make queries less consistent in ordering
- Performance
  - It should be obvious, but this will be radically different, especially for medium datasets and upwards
- Always do a final test on your target database

Optional Exercise 5.3: Using an Embedded Database

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Handling Enumeration Types by Ordinal

- MyBatis supports persisting Java `enum` types by ordinal value (position within `enum`)
  - An `EnumOrdinalTypeHandler` can be used, but must be specified explicitly

```
<resultMap type="Product" id="ProductWithTypeIdMap">
    <id     property="productId"   column="PRODUCTID"/>
    <result property="name"        column="NAME"/>
    <result property="description" column="DESCN"/>
    <result property="categoryId"  column="CATEGORY"/>
    <result property="type"        column="PRODUCT_TYPE_ID"
        typeHandler="org.apache.ibatis.type.EnumOrdinalTypeHandler"/>
</resultMap>

<select id="getProductsWithTypeId" resultMap="ProductWithTypeIdMap">
    SELECT productid, name, descn, category, product_type_id
    FROM   product
    WHERE  product_type_id IS NOT NULL
</select>

<insert id="insertProductWithTypeId" parameterType="Product">
    INSERT INTO product (productid, name, descn, category, product_type_id)
    VALUES (#{productId}, #{name}, #{description}, #{categoryId},
        #{type, typeHandler=org.apache.ibatis.type.EnumOrdinalTypeHandler})
</insert>
```

```
public enum ProductType {
   PHYSICAL_MEDIA, DIGITAL_MEDIA, HYBRID_MEDIA
}
```

Ordinal values
0, 1, 2

```
public class Product {
   private ProductType type;
   …
}
```

Custom Type Handler for `enum`

- If `enum` values in the database are not 0, 1, 2, …, define a MyBatis custom type handler
  - Implement `org.apache.ibatis.type.TypeHandler`
  - Or extend `org.apache.ibatis.type.BaseTypeHandler`

```
public enum ProductType {
    PHYSICAL_MEDIA(12), DIGITAL_MEDIA(35), HYBRID_MEDIA(44);
    private ProductType(int code) { … }
    public static ProductType of(int code) { … }
```

Codes assigned by DBA

```
public class ProductTypeHandler extends BaseTypeHandler<ProductType> {
    @Override
    public ProductType getNullableResult(ResultSet rs, String col) throws SQLException {
        return rs.getInt(col) != 0 ? ProductType.of(rs.getInt(col)) : null;
    }
    …
```

MyBatis calls this method to convert database value

Find the `enum` value that matches the database value

```
<result property="type" column="PRODUCT_TYPE_ID"
        typeHandler="com.roifmr.leap.mybatis.ProductTypeHandler"/>
```

Configure the custom type handler for this column

Discriminators

- A discriminator is a column that indicates which of a set of types is in a particular row
  - E.g., an inheritance hierarchy
- Acts as a switch statement to choose the `ResultMap`
- Child `ResultMaps` usually extend the parent map
  - If so, the map used includes all parent and child columns
  - If not, it only includes child columns

```
<resultMap id="MentorMap" type="Mentor" >
    <id property="id" column="mentor_id" />
    <result property="firstName" column="mentor_first_name" />
    <result property="lastName" column="mentor_last_name" />
    <collection … />
    <discriminator javaType="int" column="mentor_type">
        <case value="1" resultMap="FullTimeMentorMap" />
        <case value="2" resultMap="PartTimeMentorMap" />
    </discriminator>
</resultMap>

<resultMap id="FullTimeMentorMap" type="FullTimeMentor"
        extends="MentorMap">
    <result property="payPerWeek" column="pay_per_week" />
</resultMap>

<resultMap id="PartTimeMentorMap" type="PartTimeMentor"
        extends="MentorMap">
    <result property="hoursWorkedPerWeek" column="hours_per_week" />
    <result property="hourlyPay" column="pay_per_hour" />
</resultMap>
```

Passing Multiple Input Parameters

- The `parameterType` attribute specifies the type of the input parameter
  - If there's only one parameter, `parameterType` can be omitted
  - MyBatis will infer the parameter type
- Can pass multiple parameters by name
  - In the mapper interface, add `@Param` to each parameter
  - MyBatis will implicitly pass the parameter names and values in a Map
  - In `<select>`, `parameterType` is `java.util.Map`

```
<select id="getProductsByCategoryAndName"
        parameterType="java.util.Map"
        resultMap="ProductMap">
    SELECT productid, name, descn, category
    FROM   product
    WHERE  category = #{categoryId}
    AND    name LIKE #{productName}
</select>
```

```
import org.apache.ibatis.annotations.Param;

public interface ProductMapper {
  List<Product> getProductsByCategoryAndName(
              @Param("categoryId") int catId,
              @Param("productName") String name);
  …
}
```

```
@Override
public List<Product> getProducts(int catId, String name) {
  return mapper.getProductsByCategoryAndName(catId, name + "%");
}
```

Passing Multiple Input Parameters by Position

- MyBatis also supports passing multiple parameters to a mapped statement by position
  - Reference the parameters in the SELECT using `#{paramN}` syntax (*N* starts at 1)
  - No `@Param` required

```
<select id="getProductsByCategoryAndNameParam" resultMap="ProductMap">
    SELECT productid, name, descn, category
    FROM   product
    WHERE  category = #{param1}
    AND    name     LIKE #{param2} || '%’
</select>
```

```
public interface ProductMapper {
…
    List<Product> getProductsByCategoryAndNameParam(int categoryId, String name);
…
}
```

Paginated ResultSets

- Large numbers of records may be returned by some queries
- MyBatis supports pagination of large ResultSets
  - Using RowBounds
    - With offset (starting position)
    - And limit (number of records)
  - Mapper XML does not need to change
    - Add parameter to interface

```java
public interface ProductMapper {
    List<Product> getProducts();
    List<Product> getProducts(RowBounds bounds);
    …
}
```

```java
@Override
public List<Product> getProductsByBounds(int offset, int limit) {
    RowBounds bounds = new RowBounds(offset, limit);
    return mapper.getProducts(bounds);
}
```

```java
// display the third page of 25 records
List<Product> productsCurrentPage = dao.getProductsByBounds(50, 25);
```

SqlSession and Mappers

- The core of MyBatis is `SqlSession`
  - Has methods to execute SQL commands (`select`, `selectList`, `insert`, `update`, etc.)
  - Mapper methods are convenience methods mapped to `SqlSession`

- Sometimes it is useful to use `SqlSession` directly
  - When not using Spring, we might write:

```
try (SqlSession session = sqlSessionFactory.openSession()) {…}
```

  - Do NOT do this in Spring
    - That `SqlSession` is not thread-safe and will not participate in Spring transactions
  - Instead create a bean from `SqlSessionTemplate`

```
<bean id="sqlSession" class="org.mybatis.spring.SqlSessionTemplate">
    <constructor-arg index="0" ref="sqlSessionFactory" />
</bean>
```

```
http://mybatis.org/spring/sqlsession.html
```

Using a Custom `ResultHandler`

- There may be a situation where custom processing of query results is necessary
  - Define a custom `ResultHandler`
  - The `handleResult` method will be called for every row returned by the query
- E.g., to return a Map with one property as key and another (not the entire object) as value

```
@Override
public Map<Integer, String> getProductIdNameMap() {
    Map<Integer, String> map = new HashMap<>();
    session.select("com.fidelity.integration.ProductMapper.getProducts", // query
            new ResultHandler<Product>() {
                @Override
                public void handleResult(ResultContext<? extends Product> context) {
                    Product product = context.getResultObject();
                    map.put(product.getProductId(), product.getName());
                }
            });
    return map;
}
```

Calling a Stored Procedure with MyBatis

- MyBatis can call stored procedures
- Stored procedures that do not return results sets are straightforward:
  - To pass more than one parameter, use the parameter map method, or a helper class
  - The stored procedure can execute one or more INSERT, UPDATE, or DELETE
  - Parameter modes: `IN` (read only), `OUT` (write only), `INOUT` (read and write)
  - `jdbcType` is required only for `NULLABLE` values

```
<update id="deleteProductsByCategoryProcedure" parameterType="int" statementType="CALLABLE">
    { CALL proc_del_products_by_category( #{categoryId, mode=IN, jdbcType=NUMERIC} ) }
</update>
```

- You can configure a stored procedure call using annotations:

```
@Update("{ CALL proc_del_products_by_category(#{categoryId, mode=IN, jdbcType=NUMERIC}) }")
@Options(statementType = StatementType.CALLABLE)
void deleteProductsByCategory(@Param("categoryId") int categoryId);
```

Stored Procedures That Return Results

- Each vendor treats this differently, here are two examples
  - Oracle can only return results as an output parameter

```
<!-- Oracle style procedure returning a SYS_REFCURSOR as second parameter -->
<select id="getProductsByCategoryProcedure" parameterType="java.util.Map" statementType="CALLABLE">
        { CALL proc_products_by_category( #{categoryId, mode=IN, jdbcType=NUMERIC},
        #{results, jdbcType=CURSOR, mode=OUT, javaType=java.sql.ResultSet, resultMap=ProductMap} ) }
</select>
```

  - MyBatis returns them as a member of the Map

```
…
@SuppressWarnings("unchecked")
List<Product> products = (List<Product>) parameterMap.get("results");
return products;
```

  - HyperSQL can return results directly and MyBatis treats them like any other query

```
<!-- HyperSQL style procedure returning a CURSOR as a result set  -->
<select id="getProductsByCategoryProcedure" parameterType="int" statementType="CALLABLE"
            resultMap="ProductMap">
    { CALL proc_products_by_category( #{categoryId, mode=IN, jdbcType=NUMERIC} ) }
</select>
```

Optional Exercise 5.4:
Calling a Stored Procedure with MyBatis

HANDS-ON EXERCISE

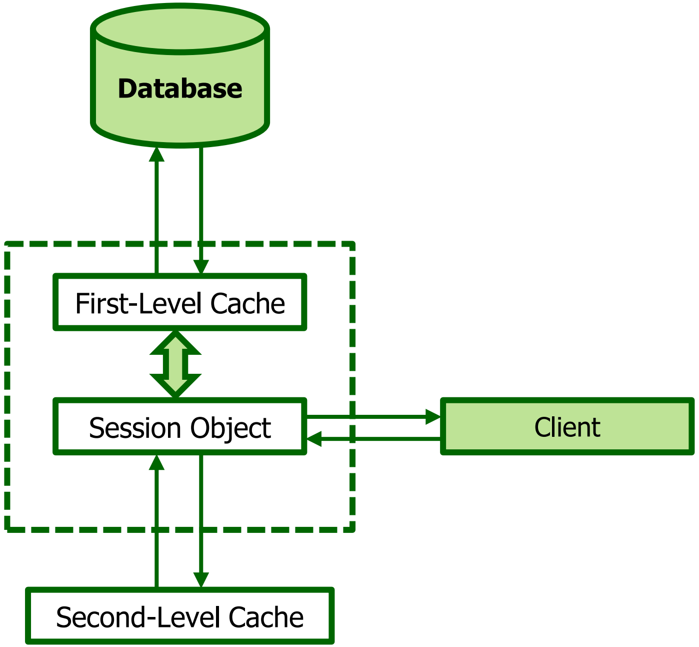
20 min

- Follow the directions in your Exercise Manual

Caching in MyBatis

- MyBatis provides support for caching query results from `<select>` statements
  - First-level cache is enabled by default
  - Short-lived: managed by a single SqlSession
  - When possible, SqlSession retrieves results from first-level cache instead of executing the same query twice

- Global second-level caches can be enabled
  - Using the `<cache/>` element in Mapper XML files
  - Results can be cached in memory or written to disk
  - May be shared by multiple SqlSessions
  - This can also be customized



Optional

Second-Level Caching in MyBatis

- Adding `<cache/>` to a Mapper XML file does the following:
  - All results from `<select>` statements will be stored in the cache
  - All `<insert>`, `<update>`, and `<delete>` statements flush the cache
  - The cache uses a Least Recently Used (LRU) policy
  - There is no flush interval
  - The cache will store up to 1024 references to lists or objects
  - The cache is a read/write cache
    - Retrieved objects are not shared
    - They can be safely modified by the caller
    - There will be no interference with other caller's modifications

```
<cache />
```

Customizing Second-Level Caching

```
<cache eviction="FIFO" flushInterval="60000" size="512" readOnly="true"/>
```

- Caching can be customized by setting attributes in the `<cache>` element
  - The eviction attribute sets the eviction policy
    - LRU, FIFO (First in first out), SOFT (soft reference), WEAK (weak reference)
  - The `flushInterval` attribute
    - Cache flush interval in milliseconds
  - The size attribute
    - The maximum number of elements stored in the cache
  - The `readonly` attribute
    - A read-only cache will return the same cached object to all callers
    - A read-write cache will return a serialized copy of a cached object

- MyBatis also integrates with third-party cache libraries
  - Like OSCache, Ehcache, Hazelcast

Optional Exercise 5.5: Caching with MyBatis

HANDS-ON EXERCISE

30 min

- Follow the directions in your Exercise Manual

Chapter Concepts

DML Through MyBatis with XML

Transaction Management in Testing

SQL Mappers Using Annotations

Advanced Topics

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Performing database update operations with MyBatis
- Managing transactions in JUnit tests with Spring
- Configuring MyBatis with annotations
- Using MyBatis custom type handlers and query caching
- Using advanced MyBatis features

Fidelity LEAP

Technology Immersion Program

Mastering Spring and MyBatis

Chapter 6: Functional Programming

Chapter Overview

In this chapter, we will explore:

- Functional programming
  - A style of structuring a computer program
  - Treats computation as the evaluation of mathematical functions
  - Avoids the use of mutable data
- Java support for functional programming
  - Functional interfaces
  - Lambda expressions
  - `Optional` variables
  - Stream API

Chapter Concepts

Functional Programming

Lambda Expressions

Stream API

Optional Variables

Chapter Summary

Functional Programming

- Many languages (Java, C++, C#) are based on *imperative programming*
  - Statements change the running state of the program
- But some languages (LISP, Haskell, Erlang) are based on *functional programming*
  - Functions map values to other values
  - Use functions and expressions (declarations) instead of statements
- In a functional language, output of a function depends only on the input arguments
  - Calling the same function twice with the same arguments returns the same value
  - No dependence on local or global state
- Eliminates side effects
  - No change in state that does not depend on the function inputs
  - No need to worry about changes that you cannot see
- The Java team started adding functional programming features in Java 8

Chapter Concepts

Functional Programming

Lambda Expressions

Stream API

Optional Variables

Chapter Summary

Lambda Expressions

- A *lambda expression* can be thought of as an anonymous function
  - Has a list of parameters, a body, and a return type
  - Can be passed as an argument to another method

- The name "lambda" comes from the *lambda calculus*
  - Developed by mathematician Alonzo Church in the 1930s
  - A system developed to study and formalize the concept of functions

- Java lambda expression: concise way to define and use a callback method
  - Can replace a full class definition

Lambda Expressions (continued)

- A lambda expression can replace an anonymous inner class

```
ticketList.sort(new Comparator<Ticket>() {
    @Override
    public int compare(Ticket t1, Ticket t2) {
        return Double.compare(t1.getCost(), t2.getCost());
    }
});
```

Defines the comparison function as an anonymous inner class

  – Here is the same operation using a lambda expression

```
ticketList.sort((t1, t2) -> Double.compare(t1.getCost(), t2.getCost()));
```

Defines the comparison function as a lambda

- A lambda expression can be used anywhere a *functional interface* is used
  – Functional interface: an interface that specifies a single abstract method

What Is a Functional Interface?

- `java.util.function` package defines standard functional interfaces
  - Example: method with a `java.util.function.Function` parameter

```
public List<String> applyFunction(List<String> items, Function<String, String> callback) {
    List<String> newItems = new ArrayList<>();
    for (String item : items) {

        String newItem = callback.apply(item);

        newItems.add(newItem);
    }
    return newItems;
}
```

Call the "function" that was passed as an argument

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Compiler enforces "single abstract method" rule

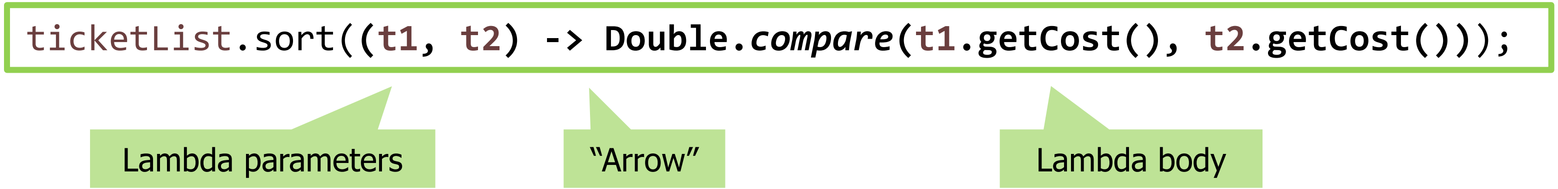
Types of `Function`'s argument and return value

- Pass a lambda as the value of the `Function`'s callback argument
  - Compiler will generate a class that implements the interface

```
List<String> nobles = List.of("Helium", "Neon", "Argon", "Krypton", "Xenon", "Radon");
List<String> lowerNobles = applyFunction(nobles, s -> s.toLowerCase());
List<String> emphaticNobles = applyFunction(nobles, s -> s.concat("!"));
```

Lambda Expressions Syntax

- A lambda expression consists of three sections
  - Lambda parameters
  - Arrow
  - Lambda body



```
ticketList.sort((t1, t2) -> Double.compare(t1.getCost(), t2.getCost()));
```

- Internally, the Java compiler converts a lambda to an anonymous inner class

Lambda Expression vs. Inner Class

- A lambda expression has some important differences from an inner class

- Inner class creates a new scope
  - Can overwrite local variables from enclosing scope
    - Instantiate a new local variable with the same name in the inner class
    - Use the keyword ***`this`*** in the inner class to refer to its instance

- Lambda expressions work with the enclosing scope
  - Cannot overwrite variables from enclosing scope in the lambda's body
  - The key word ***`this`*** refers to the enclosing instance

- However, both lambdas and anonymous inner classes can only access variables of the enclosing scope if they are "effectively final" or `final`
  - A local variable that is not changed after initialization
  - You may need to add `final` to variable or parameter declarations

Lambda Expressions – Prefer Simplest Syntax

- Let the compiler determine parameter types
  - It will tell you if there is an ambiguity
- Braces and return statements not required for one-line lambda bodies
- Parentheses not required for one parameter
- Call a helper method if the body is complex

```
// prefer this
a -> a.toLowerCase()

// to this
(String a) -> { return a.toLowerCase(); }
```

Instead of a complex lambda …

```
translateInput(s -> {
  String result = …;
  … // many lines of code
  return result;
});
```

… call a helper method

```
translateInput(s -> translateHelper(s));

private String translateHelper(String s) {
  String result = …;
  … // many lines of code
  return result;
}
```

Method References

- Many lambdas simply call an existing method

```
ticketList.forEach(t -> System.out.println(t));
```

Pass a lambda to `forEach()`

- If a lambda's only operation is to pass parameters to an existing method, it can be replaced by a *method reference*

```
ticketList.forEach(System.out::println);
```

Pass a method reference to `forEach()`

- Internally, the Java compiler converts a method reference to a lambda
- Method references may be used for static methods or instance methods
  - Syntax: `class-or-object-name::method-name`

Method References (continued)

- Example: sorting a list of tickets

Existing method in `Ticket` class

```
public class Ticket {
  public static int compareByCost(Ticket t1, Ticket t2) {
     return Double.compare(t1.getCost(), t2.getCost());
  }
  …
}
```

Lambda calls existing method

```
ticketList.sort((t1, t2) -> Ticket.compareByCost(t1, t2));
```

```
ticketList.sort(Ticket::compareByCost);
```

Replace lambda with method reference

- Method references are often used as arguments to Stream methods
  - We'll see these soon

Exercise 6.1: Lambda Expressions

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Concepts

Functional Programming

Lambda Expressions

Stream API

Optional Variables

Chapter Summary

Streams

- Java streams are designed to process collections of values easily and efficiently
- The stream library implementation manages the scheduling of the operations
  - Supports parallel operations: each stream operation could execute in its own thread
- Streams work on the “what, not how” principle
  - You describe what needs to be done
  - You don’t specify how to carry out the operations

Streams vs. Iteration

- Suppose we have a list of words
  - Goal: find all words longer than 10 characters

```
// count the long words by iterating thru the list
int count = 0;
for (String w : words) {
   if (w.length() > 10){
       count++;
   }
}
```

```
// count the long words by using streams
long count = words.stream()
                  .filter(w -> w.length() > 10)
                  .count();
```

- Best practice: use streams instead of iterating over a collection
  - More explicit than using `for` loops and `if` statements

Stream Characteristics

- Streams do not store their elements
  - They may be stored in a collection
  - Or generated on demand
- Streams do not modify their source
  - They return a new stream with the results
- Streams are lazy whenever possible
  - May not be executed until the results are needed
  - May even process an infinite stream sometimes

Streams

- A typical stream operation has three stages

1. Create the stream
2. Specify the intermediate operations
   a. Transform initial stream into other streams
3. Apply a terminal operation to produce a result
   a. This will execute any lazy operations
   b. Nothing happens until the terminal operation is called

The `stream()` method creates a stream for a list

The `filter()` method returns another stream, in this case, with words longer than 10 characters

The `count()` method reduces the stream to a `long` result (in this case, the number of words longer than 10 characters)

```
// count the long words by using streams
long count = words.stream()
                  .filter(w -> w.length() > 10)
                  .count();
```

Stream Operations with `collect()`

- Task: create a new list by performing the same operation on all items of an existing list
  - Problem: stream methods produce new streams, not lists
  - Solution: create a list from a stream using a `Collector`

- Pass a `Collector` object to the stream terminal operation `collect()`
  - `Collectors` has methods that create collector objects
  - Provides support for counting, summing, averaging, grouping, etc.

```
import java.util.stream.Collectors;
…
// convert strings to lowercase using streams
List<String> names = List.of("LoGan", "kElSEy", "SLOAN");

List<String> lowerNames =
        words.stream()
             .map(s -> s.toLowerCase())
             .collect(Collectors.toList());
```

Could use a method reference:
`.map(String::toLowerCase)`

Chapter Concepts

Functional Programming

Lambda Expressions

Stream API

Optional Variables

Chapter Summary

NullPointerException Problem

- NullPointerException (NPE) is a very common Java problem
  - Any method that returns an object reference might return a null value
  - If the caller forgets to check the method's return value, an NPE may occur

```
public Person findPerson(String name) {
    Person p = dao.queryPerson(name);
    return p;
}
…
Person person = findPerson("Pat Drie");

String email = person.getEmailAddress();
```

DAO method may return null

No test for null

If person is null, it causes an NPE

- Caller of findPerson() forgot to ask, "What should happen if I get a null value?"

java.util.Optional Class

- Java introduced the `Optional` class to prevent NPEs
- Use `Optional` as a method's return value
  - A method that returns `Optional` cannot return `null`
  - Caller of that method must call an `Optional` method to get the "real" value

```
public Optional<Person> findPerson(String name) {
    Person p = dao.queryPerson(name);
    return Optional.ofNullable(p);
}
...
Person person = findPerson("Pat Drie").orElse(new Person("", ""));

String email = person.getEmailAddress();
```

Method wraps the `Person` in an `Optional`

`Optional` always yields a non-null object

NPE is impossible

- `Optional` is a "speed bump"—it forces developers to slow down and think
  - "What should happen if I get an `Optional` with no value?"

Optional Conditional Methods

- Optional can replace explicit tests for null values
- Without Optional:

```
Person person = findPerson(name);
if (person != null) {
    userService.add(person);
}
```

Explicit test for null

Execute the callback only if the Optional has a value

- With Optional:

```
findPerson(name).ifPresent(p -> userService.add(p));
```

```
findPerson(name).ifPresentOrElse(p -> userService.add(p),
                                () -> logger.info(name + " not found"));
```

Execute the second callback if the Optional has no value

- Optional defines methods that may be chained: filter(), map()
- Best practice: return Optional from methods that return object references

Chapter Concepts

Functional Programming

Lambda Expressions

Stream API

Optional Variables

Chapter Summary

Exercise 6.2: Working with Streams and `Optional`

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Summary

In this chapter, we have explored:

- Functional programming
  - A style of structuring a computer program
  - Treats computation as the evaluation of mathematical functions
  - Avoids the use of mutable data
- Java support for functional programming
  - Functional interfaces
  - Lambda expressions
  - `Optional` variables
  - Stream API

Key Points

Pattern/Principle	Pointers
Lambda expressions	• Favor expressions over statements • Chain lambda expressions instead of growing them
Functional interfaces	• Define one abstract method per interface • Use the `@FunctionalInterface` annotation
Optional variables	• Use `Optional` as a return value • Do not use `Optional` as a field or argument • Use *`orElse()`* instead of *`get()`*
Streams	• Prefer streams for iterating over collections • Style favors intention over process

Fidelity LEAP

Technology Immersion Program

Mastering Spring and MyBatis

Course Summary

Course Summary

In this course, we have:

- Used the Spring framework to build clean, extensible, loosely-coupled enterprise Java applications
- Utilized Spring as an object factory and dependency injection to wire components together
- Understood and applied MyBatis to simplify access to relational databases
- Explored and applied Spring to simplify the use of MyBatis in an application
- Applied transaction strategies via configuration