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

Course Objectives

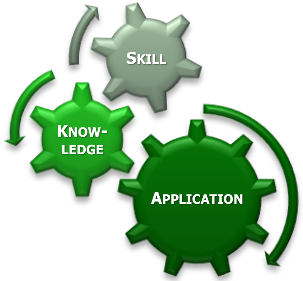
In this course, we will:

- 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

Annotation-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

Using Spring's Object Factory

- Consider the following simple plain Java code
  - The `PopupGreeter` class has a dependency on the `Visitor` interface
    - An interface is used to maintain loose coupling between greeter and visitor

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

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

    @Override
    public void setVisitor(Visitor v) {
        this.visitor = v;
    }
}
```

Has dependency on Visitor

```java
public interface Greeter {
    public void greet();
    public void setVisitor(Visitor v);
}
```

```java
public interface Visitor {
    public String getName();
    public String getGreeting();
}
```

Using Spring's Object Factory (continued)

- `AmarilloVisitor` is a JavaBean providing the following properties
  - A read and write property called `name`
  - A read-only property named `greeting`
- Remember, property names are defined by the name of the get/set methods
  - Not the name of the data member they access

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

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

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

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

    public void setName(String name) {
        this.name = name;
    }
}
```

Using Spring's Object Factory for the `PopupGreeter`

- To use the `PopupGreeter` class with the `AmarilloVisitor`, we need to:
  - Create an instance of `PopupGreeter`
  - Create an instance of the `AmarilloVisitor`
    - Call `setName` on the visitor passing in a `String`
    - Call `setVisitor` on `PopupGreeter` passing the instance of `AmarilloVisitor`
- Spring's object factory will create the two objects for us – IoC
  - Pass a name configured to `setName()` of the `AmarilloVisitor` – DI
  - And pass the `AmarilloVisitor` to `setVisitor()` of the `PopupGreeter` – DI
- To use Spring, we need to:
  - Configure the factory
  - Instantiate the factory and ask it for the `PopupGreeter`
- The Spring factory creates objects referred to as Spring-Managed Beans
  - JavaBeans that may not have a no-arg constructor (although they often do)

Spring Factories: `ApplicationContext`

- Spring provides several factories that may be used by an application
  - All of them implement the `ApplicationContext` interface
  - `org.springframework.context.ApplicationContext`
  - Configuration is provided in XML file(s) or by annotations or both
- Any `ApplicationContext` implementation is capable of:
  - Instantiating a bean (IoC)
  - Injecting its dependencies (DI)
  - Providing access to that bean to any code that requires it
- Each factory is its own “container”
  - Manages the lifecycle of the beans that it creates
  - The factory constructs beans as needed
    - Can also dispose of beans when the factory is shut down
- Objects created by factory are known as *Spring-managed beans*

ClassPathXmlApplicationContext

- This factory expects the configuration in XML file(s)
  - File(s) should be present somewhere in the classpath:

Should be on the classpath

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

// Use factory

factory.close();
```

- The `close()` method is not available in the `ApplicationContext` interface
  - So, we declare the factory as `AbstractApplicationContext`
  - Declare variables as the most general type that gives the functionality you need

Using Beans from Factory

- Beans are provided by the `getBean` methods on `factory`
  - An id/name is provided for each bean when configured
    - Id is used when asking for the bean from the factory

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

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

factory.close();
```

Use bean

Id of bean requested

Interface bean should implement

XML Configuration File

- XML configuration file has `<beans>` root element
- Every bean to be created is configured using `<bean>` element
  - Id is provided to enable access by clients requesting bean from factory
  - Fully qualified class name so Spring knows which type of object to create

```
<?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="vis" class="com.fidelity.greeter.AmarilloVisitor" />

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

Id of bean when requested from factory

Configuring Dependency Injection in XML

- Spring will automatically call setter methods on properties if configured
- Values passed to properties can be:
  - Other Spring created objects (beans)
  - Absolute values configured in XML

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

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

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

Call `setName` on `AmarilloVisitor` with value as parameter

Call `setVisitor` on `PopupGreeter` with bean whose id is `vis` as parameter

Exercise 1.1: A Simple Spring Application

HANDS-ON EXERCISE

30 min

- Follow the directions in your Exercise Manual

How Many Beans Will Be Created?

- The `getBean` method may:
  - Keep returning the same object each time
    - A "singleton"
    - One bean per Spring container (`ApplicationContext` instance)
  - Return a freshly instantiated object each time
    - Uses the XML 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
  - Five scopes are available
    - Singleton and prototype
    - Request and session for web applications
    - Global session for Portlets

We will use only these two scopes

Bean Scope Example

- The scope attribute of the `<bean>` element is used to define the scope
  - If omitted, singleton is the default

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

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

New bean per request from factory

Chapter Concepts

The Spring Object Factory

Annotation-Based Factory Configuration

Chapter Summary

Configuration Using Annotations

- It is possible to use annotations for configuration rather than XML
- To use annotation-based configuration requires:
  1. Enable annotation-based configuration in XML configuration file
  2. Annotating any class(es) that are to be Spring-managed beans
- `@Component` is used to mark a class as a Spring-managed bean
  - Also `@Service`, `@Repository`, `@Controller` can be used
- Dependency injection is supported
  - Use `@Autowired` or `@Inject`
  - Both annotations work the same way
- `@Autowired` is Spring-specific
  - `@Inject` is a Java JEE standard annotation
    - Needs JEE jar file in classpath to use this

Annotation Configuration in XML

- In the XML configuration file, tell Spring to scan the package(s) that contain(s) the bean

```
<?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">
    <description>PopupGreeter driven by annotations</description>

    <context:annotation-config/>

    <context:component-scan base-package="com.fidelity.greeter" />
</beans>
```

Scan XML defined classes for annotations

Scan for annotated beans

Can be comma-separated list

- Only need one of these because `component-scan` also includes `annotation-config`
  – `annotation-config` allows XML defined classes to have annotated properties
  – `component-scan` allows beans to be defined with annotations (e.g., `@Component`)

Annotation Example

- Annotate Spring-managed beans:

Bean id is defined by value passed to annotation

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

    @Autowired
    public void setVisitor(Visitor visitor) {
        System.out.println("visitor set to " + visitor);
        this.visitor = visitor;
    }
    // etc
}
```

- `@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 bean implements `Visitor`

`@Qualifier` and `@Primary`

- Need `@Qualifier` if there could be multiple beans of required type

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

    @Autowired
    @Qualifier(“dehliVis")
    public void setVisitor(Visitor visitor) {
        System.out.println("visitor set to " + visitor);
        this.visitor = visitor;
    }

    // etc
}
```

Id of bean to inject

- Or annotate one of the definitions with `@Primary`
  - Can use `primary=true` in XML

Annotation Stereotypes

- When configuring a class using annotations the following annotations can be used:
  - `@Component`
  - `@Controller`
  - `@Service`
  - `@Repository`
- A Component is a catch-all
  - Controllers are associated with the presentation tier
  - Services are associated with the business tier
  - Repositories are associated with the integration tier
- Can also specify the scope using `@Scope`
  - As with XML, default is singleton

```
@Component("greeter")
@Scope(value="prototype")
public class PopupGreeter implements Greeter {
```

Injecting a Value with Annotations

- Value injection is available using annotations also
  - But pointless, since we can just as easily hardcode the value!

```
@Value(value="Bruce")
public void setName(String name) {
    this.name = name;
}
```

  - Will see better use of `@Value` when we discuss the Spring expression language

Exercise 1.2: Spring with Annotations

HANDS-ON EXERCISE

30 min

- Follow the directions in your Exercise Manual

Complete Configuration Using Annotations

- XML can be completely removed from a Spring application
- Spring allows for a class to be annotated with `@Configuration`
  - Creation of objects is completed in code
  - Spring calls this “Java Configuration”

```
@Configuration
public class AppConfig {
    @Bean
    public Visitor createVisitor(){
        return new AmarilloVisitor();
    }

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

Annotation-Based Factory Creation

- The Configuration class can be used to create an `ApplicationContext`
  - Use `AnnotationConfigApplicationContext`

```java
AbstractApplicationContext factory =
        new AnnotationConfigApplicationContext(AppConfig.class);

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

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

g.greet();

factory.close();
```

Exercise 1.3: Java Configuration

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Concepts

The Spring Object Factory

Annotation-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 XML or annotations (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

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

```
<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 equivalent code:

```
Visitor vis = new com.fidelity.greeter.AmarilloVisitor();
// Any setter injection defined for vis

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

Equivalent Java Code

- Helpful to map XML configuration with equivalent Java code

When Spring Sees …	Equivalent Java Code
`<bean id="x" class="a.b.C">`	`x = new a.b.C();`

`<bean id="x" class="a.b.C">`
   `<property name="`**z**`" ref="y">`
	`y = …; // configured earlier`
	`x = new a.b.C();`
	`x.`**setZ**`(y);`

p-namespace

- Spring provides a shortcut of the following standard approach:

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

- Using the p-namespace:

```
<beans …
       xmlns:p="http://www.springframework.org/schema/p"
       … >

    <bean id="vis" class="com.fidelity.greeter.AmarilloVisitor"
        p:name="Joe Bob Springstein" />

    <bean id="greeter" class="com.fidelity.greeter.PopupGreeter"
        p:visitor-ref="vis" />
</beans>
```

- The above two configurations are functionally equivalent

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
  - Need `@Qualifier` or `@Primary` if there could be multiple beans of required type

Injecting a Value

- The IoC container can inject many types of *values*
  - Not just other beans
  - Although beans are the most common

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

- Can inject primitives and Strings ("values"):

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

- Value injection is available using annotations also
  - But pointless, since we can just as easily hardcode the value!

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

Equivalent Java Code

- Helpful to map XML configuration with equivalent Java code

When Spring Sees …	Equivalent Java Code
`<bean id="x" class="a.b.C">`	`x = new a.b.C();`

`<bean id="x" class="a.b.C">`
`   <property name="z" ref="y">`
	`y = …; // configured earlier`
	`x = new a.b.C();`
	`x.setZ(y);`

`<bean id="x" class="a.b.C">`
`   <property name="z" value="5">`
	`x = new a.b.C();`
	`x.setZ(5);`

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 configured like this:

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

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

  – The BeanFactory uses the default constructor to create object

- The `AmarilloVisitor` and `PopupGreeter` classes must have no-arg constructors
  – The fields are set by a setter method, not in constructor

Constructor Arguments

- What if a bean has no default constructor?

```
public class Actor {
    private String firstName;
    private String lastName;

    public Actor(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
```

- Need to inject a constructor argument
  - If the constructor has multiple arguments, Spring will attempt to match by type
    - If different arguments have same type, then configuration error
    - Safer to explicitly provide index of argument

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

- The `constructor-arg` element behaves just like the `property` element
  - E.g., can accept value, ref or collections (see later)

Annotation-Based Construction

- The @Autowired annotation supports constructors also

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

- Can apply @Qualifier to the constructor parameters
  – 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;
    }
…
```

Equivalent Java Code

- Helpful to map XML configuration with equivalent Java code

When Spring Sees …	Equivalent Java Code
`<bean id="x" class="a.b.C">`	`x = new a.b.C();`

`<bean id="x" class="a.b.C">`
`<property name="z" ref="y">`
	`y = …; // configured earlier`
	`x = new a.b.C();`
	`x.setZ(y);`

`<bean id="x" class="a.b.C">`
`<property name="z" value="5">`
	`x = new a.b.C();`
	`x.setZ(5);`

`<bean id="x" class="a.b.C">`
`<constructor-arg ref="y">`
	`y = …; // configured earlier`
	`x = new a.b.C(y);`

Exercise 2.2: Dependency Injection with Constructors

HANDS-ON EXERCISE

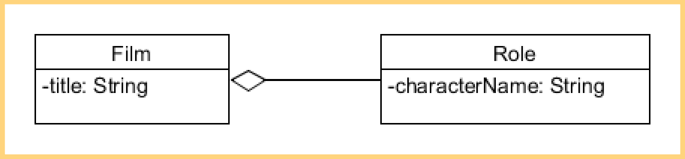
20 min

- Follow the directions in your Exercise Manual

Fully Configured Beans

- Spring will not inject a bean unless the bean is completely configured
  - `Film` depends on a list of `Role` beans

```
<bean id="no-way" class="com.fidelity.circular.Film">
    <constructor-arg index="0">
        <list>
            <ref bean="parrish" />
            <ref bean="gump" />
        </list>
    </constructor-arg>
    <constructor-arg index="1" value="Lawrence Gump and the Pirate Wall of Quantico" />
</bean>

<bean id="parrish" class="com.fidelity.circular.Role">
…
</bean>
```

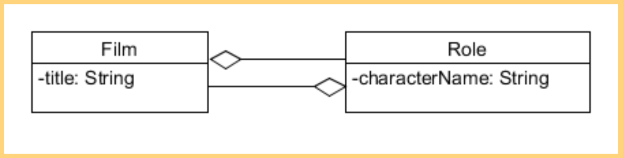


  - The `Role` beans will have to be completely configured before they are injected as a constructor argument to `Film`
    - All their dependencies will have been injected

Circular Dependency

- What if `Role` requires a reference to the `Film`?

```
<bean id="no-way" class="com.fidelity.circular.Film">
    <constructor-arg index="0">
        <list>
            <ref bean="parrish" />
…
</bean>

<bean id="parrish" class="com.fidelity.circular.Role">
    <constructor-arg index="0" ref="no-way" />
    <constructor-arg index="1" value="Alex Parrish" />
    <constructor-arg index="2" ref="chopra" />
</bean>
```



- Can't create `Film` because its constructor needs a fully configured `Role`
  – Can't create `Role` because it needs a `Film`

- Circular dependency
  – Spring will throw exception (`BeanCurrentlyInCreationException`)

Resolving Circular Dependencies

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

```
<!-- This bean does injection by setter, which breaks the circular problem -->
<bean id="no-way" class="com.fidelity.circular.Film">
    <property name="cast">
        <list>
            <ref bean="parrish" />
…
</bean>

<bean id="parrish" class="com.fidelity.circular.Role">
    <constructor-arg index="0" ref="no-way" />
    <constructor-arg index="1" value="Alex Parrish" />
    <constructor-arg index="2" ref="chopra" />
</bean>
```

- Spring will cache beans hoping to resolve circular dependencies

Constructor or Setter?

- When designing beans, you have two choices
  - Provide setters for dependencies
  - Require constructor arguments for dependencies
- Which one do you choose?

The Extremes

- The Spring team used to recommend using setters exclusively, many Spring developers adopted this approach
  - More flexible
  - Prevents circular dependencies
  - Possible that configuration may forget to set some dependency
    - Runtime `NullPointerException` when trying `access` field

- The Spring team now prefers constructors. OO purists recommend using constructors exclusively.
  - Guaranteed that all beans are valid
  - Potential for circular dependencies

When to Use Constructors and Setters

- Use constructors 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 setters for:
  - Cases where a circular dependency results
  - Mutable objects and dependencies
  - Optional dependencies
- Decide on a dependency-by-dependency basis
  - No one size fits all
  - Can have beans some of whose dependencies are injected during construction and others using setters

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`
  - Provide (mock) dependencies through constructor or setters
- Spring supplies test support classes to help mimic “Spring-like” behavior
  - E.g., mocks for `Environment` and `PropertySource`
  - `ReflectionTestUtils` can set fields or invoke methods, even private or protected

```
@Service
public class ImportantService {
    @Autowired
    private StringProvider sp;


…
```

```
ImportantService service;

@BeforeEach
void setUp() {
    StringProvider sp = new StringProvider();
    service = new ImportantService();
    ReflectionTestUtils.setField(service, "sp", sp);
}
```

- 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`

- There is an easy way to test Spring applications with JUnit:

1. Annotate the test class with `@ExtendWith(SpringExtension.class)` annotation
   - In JUnit4, was `@RunWith(SpringJUnit4ClassRunner.class)`
   - This uses a JUnit runner that creates a special test `ApplicationContext`
2. Annotate the test class with the `@ContextConfiguration` annotation
   - Pass in `beans.xml` file
   - If using a `@Configuration` class, pass it in as a parameter called `classes`
3. Declare any Spring managed beans (typically Service or DAO) being tested as fields
   - Annotate the fields with `@Autowired`
4. In the `@Test` methods, bean(s) is/are fully configured

Using the Spring TestContext Framework

- Instead of this:

```
class ImportantServiceGetBeanTest {
    ImportantService service;
    AbstractApplicationContext context;

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

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

- Use this:

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration("classpath:beans.xml")
class ImportantServiceSpringTest {
    @Autowired
    ImportantService 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` is much less common; it stores data in sort order

- Useful methods of `HashMap`:
  - `put()` allows you to add items to the map
  - `get()` allows you to obtain items to the map
    - Is actually a search operation

- `HashMaps` commonly used to cache data
  - Such as from the database or files

Using `HashMap`: An Example

- Employee is the key; Phone 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 Objects such as `Integer`
- Keys must not be null
- For reliable performance, the Key class should override `hashCode()` and `equals()`
  - The `hashCode` is usually computed from all the fields of an object

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>`
- `entrySet()`
  - Returns `Set<Map.Entry<K,V>>` containing the mappings in the map
- `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

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
  - If the Java code uses generics, Spring will honor typesafety

```
<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>
```

```
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
  - If a collection is annotated with `@Autowired`, ***all*** matching beans will be injected

Injecting Maps

- Can inject a `java.util.Map` of key-value pairs

```
<bean id="no-way-jose" class="com.fidelity.dependency.FilmWithMap">
    <property name="cast">
        <map>
            <entry key="Alex Parrish" value-ref="chopra" />
            <entry key="Forrest Gump" value-ref="hanks" />
            <entry key="Lin Mae" value-ref="jing" />
        </map>
    </property>
</bean>
```

```
public void setCast(Map<String, Actor> cast) {
    this.cast = cast;
}
```

  – Can also put `value` elements instead of `value-ref`

- Again, no direct equivalent with annotations
  – Annotating a map with `@Autowired` will cause all matching beans to be injected with a key value of the bean name

Using Properties in `beans.xml`

- You can define property values in an external file
  - Avoids need to edit `beans.xml` when changing the values

```xml
<!-- Read the Oracle database credentials from the db.properties file -->
<bean id="propertyConfigurer"
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>classpath:db.properties</value>
        </list>
    </property>
</bean>
```

```xml
<!-- Define a DataSource for the Oracle 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>
```

Using Properties in `@Configuration`

- You can do the same using annotations

```
@Configuration
@PropertySource("classpath:filmtitle.properties")
public class AppConfig {
    // Enable ${...} in @Value annotations
    @Bean
    public static PropertySourcesPlaceholderConfigurer propertyConfig() {
        return new PropertySourcesPlaceholderConfigurer();
    }
…
}
```

```
@Component("no-way")
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;
    }
…
}
```

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, map, set, properties

Fidelity LEAP

Technology Immersion Program

Mastering Spring and MyBatis

Chapter 3: Advanced Spring Configuration

Chapter Overview

In this chapter, we will explore:

- Bean lifecycle
- How dependencies are injected
- Using expressions to instantiate beans
- Managing Spring configuration across multiple files
- Debugging Spring configuration problems

Chapter Concepts

Managing Bean Lifecycle

Expression Language

More Configuration Options

Debugging Spring Configuration Problems

Chapter Summary

Specifying an Initialization and Destroy Method

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

- When a bean is configured:
  - Can specify a `destroy-method`
    - To dispose of resources when bean is no longer required
  - Can also specify an initialization method (`init-method`)
    - If object requires post-constructor initialization

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

`PostConstruct` and `PreDestroy`

- Can specify the `init` and `close` methods using annotations:

```
@Repository("dao")
public class DepartmentDao {

    // Call this method after the bean is fully configured
    @PostConstruct
    public void init() {
        …
    }

    // Call this method when we no longer need the bean
    @PreDestroy
    public void close() {
        …
    }
…
```

Ensuring Graceful Shutdown

- For desktop Java applications:
  - The `ApplicationContext` needs to register a shutdown hook with JVM
  - The method is not available from `ApplicationContext` interface
    - Only from `AbstractApplicationContext`

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

Ensure factory is automatically closed when JVM closes

- In JEE environments (such as web applications), this is not needed
  - Done automatically by web application contexts

Exercise 3.1: Bean Destroy

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Concepts

Managing Bean Lifecycle

Expression Language

More Configuration Options

Debugging Spring Configuration Problems

Chapter Summary

Spring Expression Language

- Possible to place Java runtime code in the XML file
  - `#{ expression }`
  - The `T` in an expression indicates that this is a Java type (i.e., static method)

```xml
<bean id="rand1" class="com.fidelity.advanced.RandomSimulator">
    <property name="seed" value="#{ T(java.lang.Math).random() * 100.0 }" />
</bean>
```

  - Or create new Java object and use that

```xml
<bean id="rand1" class="com.fidelity.advanced.RandomSimulator">
    <property name="seed" value="#{ new java.util.Random().nextDouble() * 100.0 }" />
</bean>
```

- Can use properties of Spring beans by chaining them

```xml
<bean id="rand2" class="com.fidelity.advanced.RandomSimulator">
    <property name="seed" value="#{ rand1.seed }" />
</bean>
```

System Properties and Environment

- Can inject system properties and environment variables directly into beans using expression language

```
<bean id="props" class="com.fidelity.advanced.SystemProperties">
    <property name="country" value="#{ systemProperties['user.country'] }" />
    <property name="winDir" value="#{ environment['SystemRoot'] }" />
</bean>
```

- All expressions also work in annotations

```
@Value("#{ systemProperties['user.country'] }")
public void setCountry(String country) {
    this.country = country;
}

@Value("#{ environment['SystemRoot'] }")
public void setWinDir(String winDir) {
    this.winDir = winDir;
}
```

Chapter Concepts

Managing Bean Lifecycle

Expression Language

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
- When using the Spring TestContext Framework, may want a special configuration
  - Different dependencies (esp. mocks)
  - Do not want to duplicate unchanged dependencies
  - Do not want to “pollute” production configuration with test
- Everywhere that accepts a configuration file or class, 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

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

  - Can also supply a list of packages in Strings
  - Or a list of classes (will scan 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

A suitable bean will automatically be injected here

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

- Also, to create two different beans from the same class

@Bean Methods (continued)

- Use @Bean when you do not own the class being instantiated
  - Cannot add annotations
- Many provided Spring classes fall into this category
  - One example is the PropertySourcesPlaceholderConfigurer mentioned earlier

```
@Configuration
@PropertySource("classpath:filmtitle.properties")
public class AppConfig {
    // Enable ${...} in @Value annotations
    @Bean
    public static PropertySourcesPlaceholderConfigurer propertyConfig() {
        return new PropertySourcesPlaceholderConfigurer();
    }
…
}
```

Using Spring Expressions for Conditional Configuration

- You can use expressions to manage which properties files are used
  - This expression reads a properties file based on environment variable `target`
    - It defaults to `prod.properties`

```
<bean id="propertyConfigurer"
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>classpath:#{environment['target'] ?: 'prod'}.properties</value>
        </list>
    </property>
</bean>
```

Chapter Concepts

Managing Bean Lifecycle

Expression Language

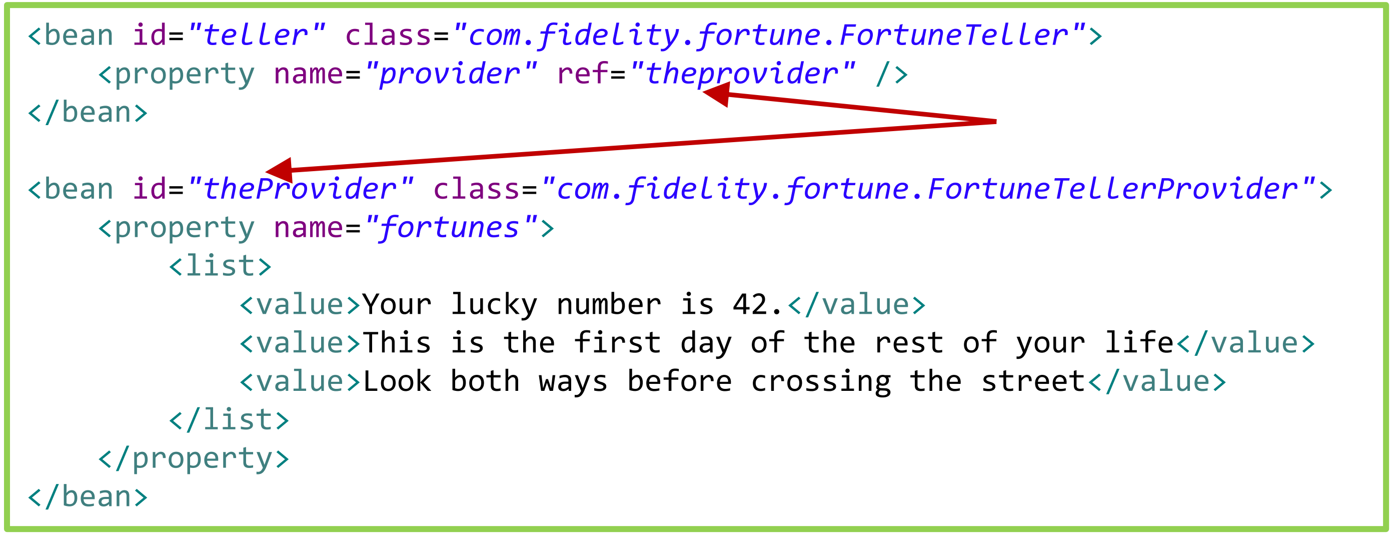
More Configuration Options

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 that bean `id` and `ref` are case sensitive



```xml
<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 log4j 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
    - Most frameworks support multiple, or use slf4j as a façade
    - Set the rootLogger to `DEBUG` level
    - You will see an astounding level of detail

```
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 Bean Lifecycle

Expression Language

More Configuration Options

Debugging Spring Configuration Problems

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Bean lifecycle
- How dependencies are injected
- Using expressions to instantiate beans
- 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

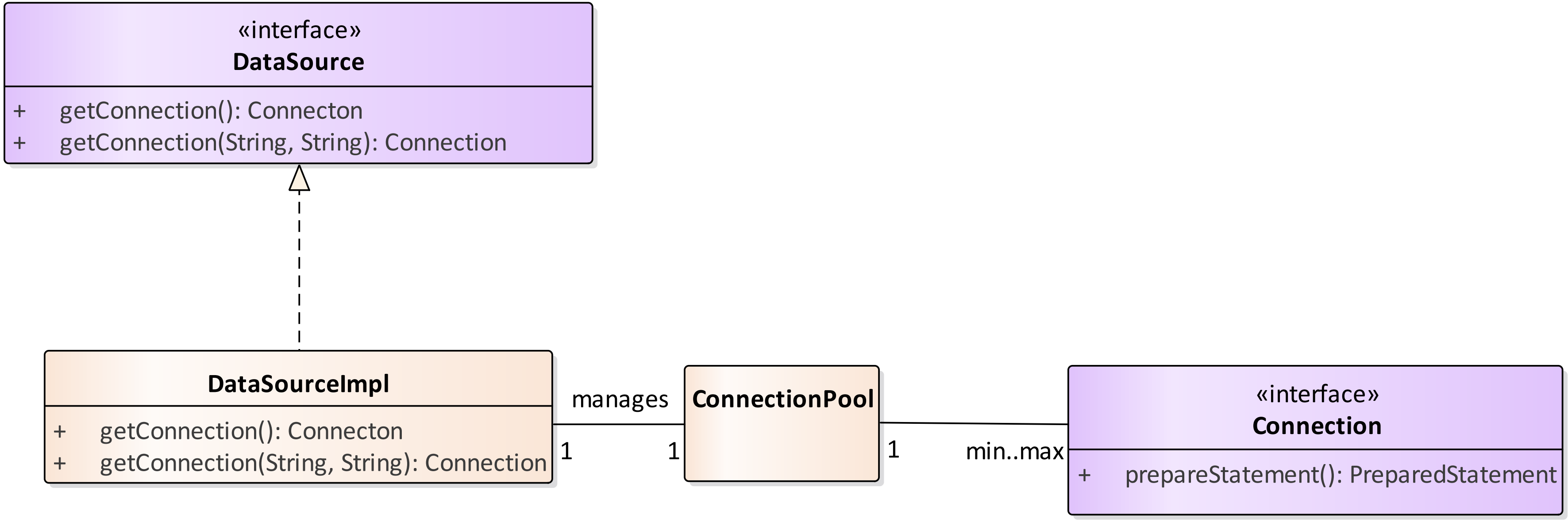
Working with Relationships

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

db.properties

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

beans.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>
```

Chapter Concepts

Configuring a DataSource

Domain Store Design Pattern

Configuring MyBatis with Spring

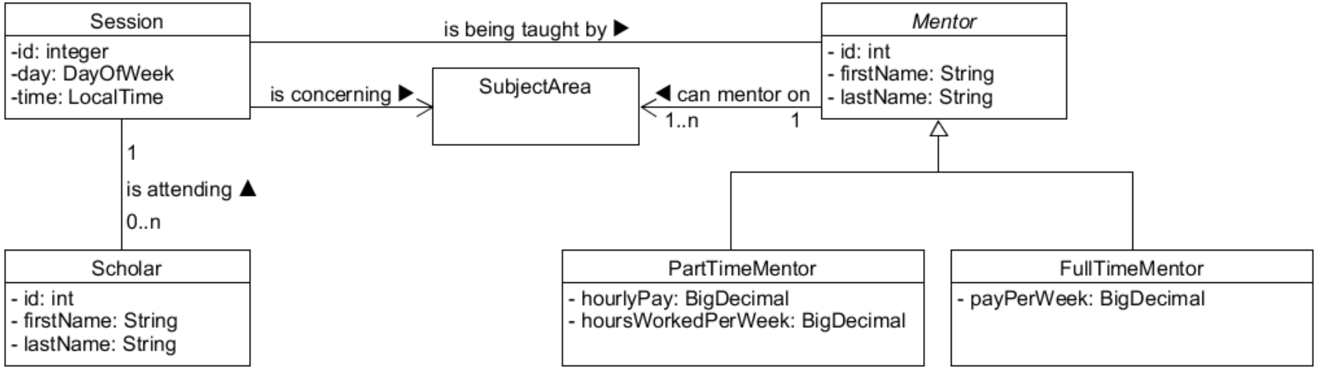
Querying a Database with MyBatis in Spring

Working with Relationships

Chapter Summary

Limitations of JDBC-Based DAOs

- Consider the YOUth Mentoring example from the Capstone project and let's focus on Session



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

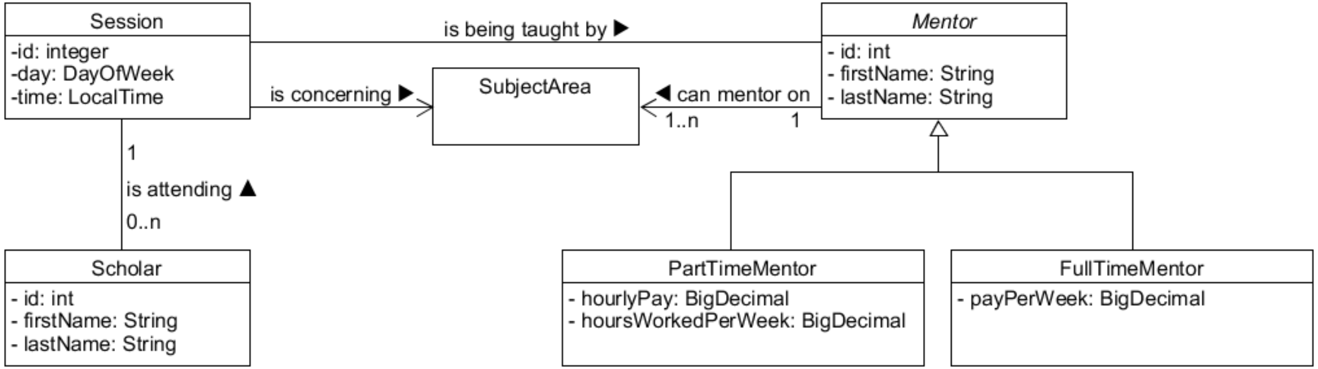
FullTimeMentor
- payPerWeek: BigDecimal

  - Each 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)



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

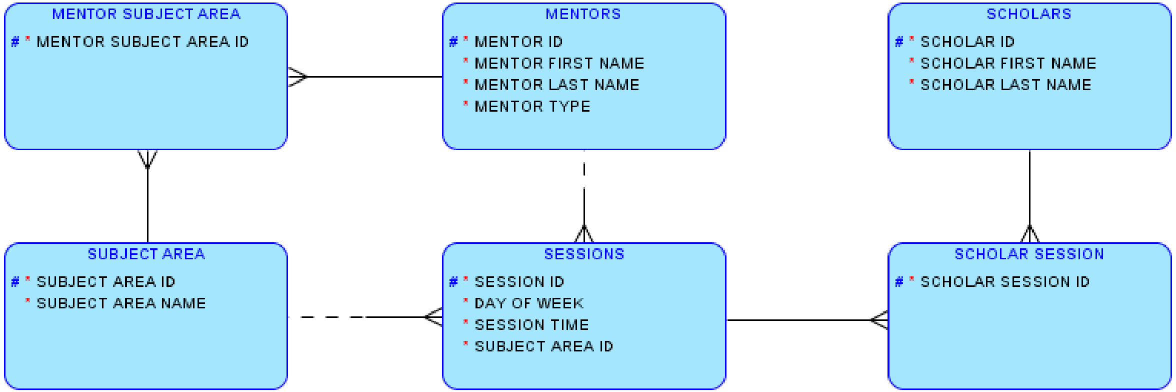
```
public abstract class Mentor {
    private int id;
    private String firstName;
    private String lastName;
    private Set<SubjectArea> subjectAreas;
…
```

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

- How can we represent these in a database?

A Complex Object Graph (continued)

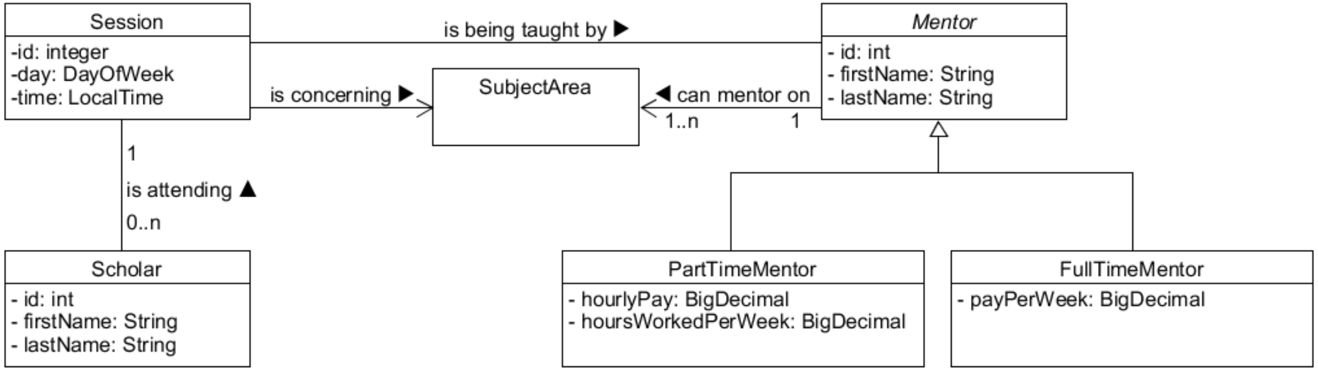
- Here is one possible representation
  - Note the tables resolving many-to-many relationships between:
    - Mentors and Subject Area
    - Sessions and Scholars



- 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?

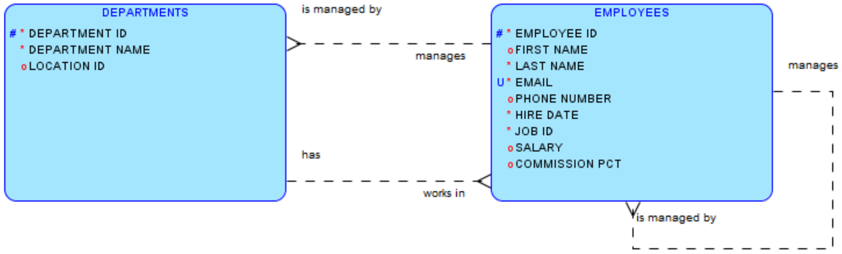
Three Common Approaches to Mapping Inheritance

- 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

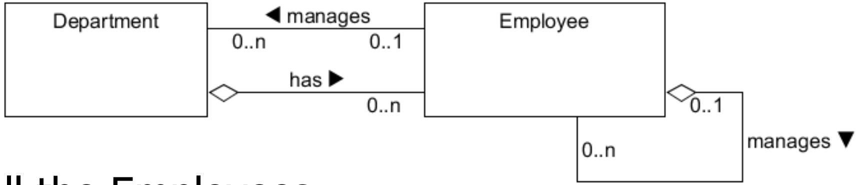
*These names are from Fowler, *Patterns of Enterprise Application Architecture*

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 an ORM 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”)
  - Update several tables when business objects are changed (“cascading”)
  - Avoid reloading data when database changes rarely (“caching”)

ORM Framework

- Use object-relational mapping (ORM) framework to persist complex object graphs
  - Instead of writing custom DAO code
  - Called the Domain Store design pattern
- Object-relational mapping frameworks provide:
  - Lazy loading
  - Cascading
  - Caching
- Examples of ORM frameworks:
  - Java Persistence API
    - Java EE Standard persistence specification
    - Several implementations available (Hibernate is one)
    - Uses its own query language
- MyBatis is not a full-fledged ORM, but it:
  - Simplifies JDBC access code through a mapping file
  - Gives full power of SQL

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

MyBatis

- MyBatis is an open-source data mapping framework
  - Maps parameters and results of SQL statements to Java objects
    - Most ORMs map Java types (BigInteger, String, etc.) to SQL types (decimal, character, etc.)
  - Provides full power of SQL
  - Can be used independently of Spring

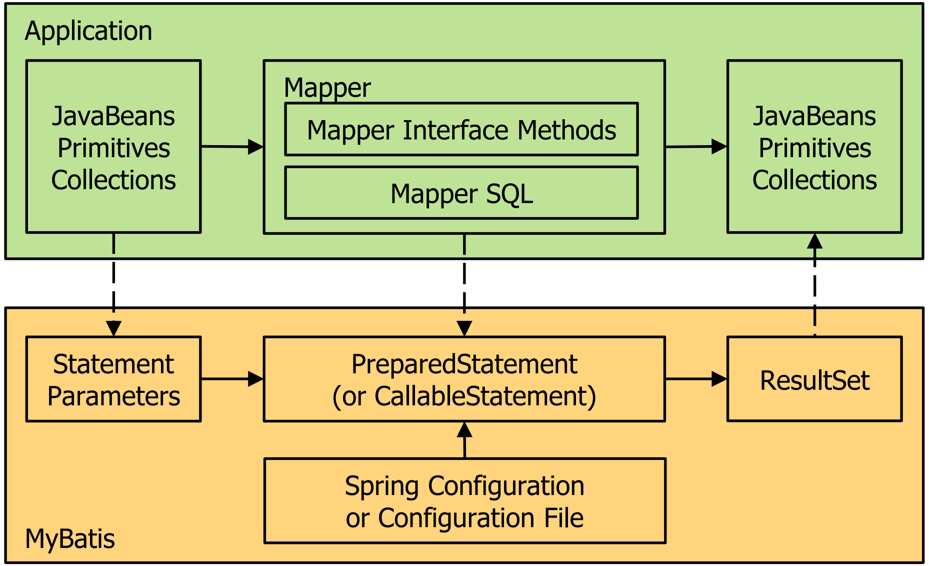
- MyBatis is a good choice choice if:
  - Performance is critical
  - Your business objects are relatively simple
    - Not too many association or inheritance relationships
  - Your application is based mostly around database stored procedures

- Hibernate/JPA may be better choice for very complex object graphs
  - User specifies mapping of a Java object to a relational data model
  - Hibernate Query Language (HQL) translated to optimal SQL "dialect"

https://github.com/mybatis/

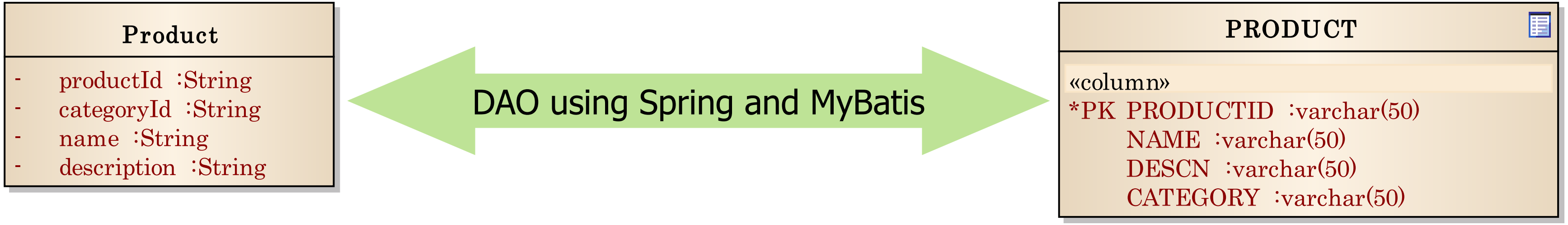
How MyBatis Works

- 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 & 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
3. Create a mapping file interface that declares the database operations
4. Write mapping file for the Java Bean
5. Use MyBatis from Service or DAO

Step 1: Configure MyBatis

- Tell Spring to do component scanning for services and autowiring in the Spring configuration file `application-context.xml`
- Specify the Java package for Spring to scan for Mapper classes
  - For autowiring into the Service or DAO beans

beans.xml

```
<!-- enable component scanning and autowire
    (beware that this does not enable mapper scanning!) -->
<context:component-scan base-package="com.fidelity.service, com.fidelity.integration" />

<!-- scan for mappers and let them be autowired -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.fidelity.integration" />
</bean>
```

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
  - Specify location of the MyBatis config file, if required
  - Specify data source to use
  - Specify the location for mapper files
  - Specify the Java package that needs to be scanned to create aliases for JavaBeans

beans.xml

```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="WEB-INF/mybatis-config.xml" />
    <property name="typeAliasesPackage" value="com.fidelity.domain" />
</bean>
```

Often do not need this when using MyBatis with Spring – can leave it out

MyBatis Configuration File

- By default, this file is named `mybatis-config.xml`
  - When using MyBatis standalone, this defines parameters such as the data source, mapper locations, and type aliases that are handled by the `SqlSessionFactoryBean`
- 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 with MyBatis
  - Provides template classes for MyBatis
    - 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 use cases using AOP
    - Application code will not explicitly begin, commit, or rollback 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**
3. Create a mapping file interface that declares the database operations
4. Write mapping file for the Java Bean
5. Use MyBatis from Service or DAO

```java
public class Product implements Serializable {
  private static final long serialVersionUID = -7492639752670189553L;

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

// getters and setters
}
```

MyBatis will use the default constructor and getter/setter methods (like Spring, can also work with constructors)

Step 3: MyBatis Mapping Interface

1. Configure MyBatis to load mapping files and use a DataSource
2. Write a Java bean
3. **Create a mapping file interface that declares the database operations**
4. Write mapping file for the Java Bean
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 4a: MyBatis Mapping File

1. Configure MyBatis to load mapping files and use a DataSource
2. Write a Java bean
3. Create a mapping file interface that declares the database operations
4. **Write mapping file for the Java Bean**
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 4b: MyBatis Mapping File

- By convention, the name of mapping file is `ClassNameMapper.xml`

`ProductMapper.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>
```

Step 4c: Mapping File Location

- Place the 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
3. Create a mapping file interface that declares the database operations
4. Write mapping file for the Java Bean
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

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

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

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

- Rather than use column aliases, we can use a `ResultMap` to define the mapping
  – `ResultMaps` allow more complex mappings to be defined

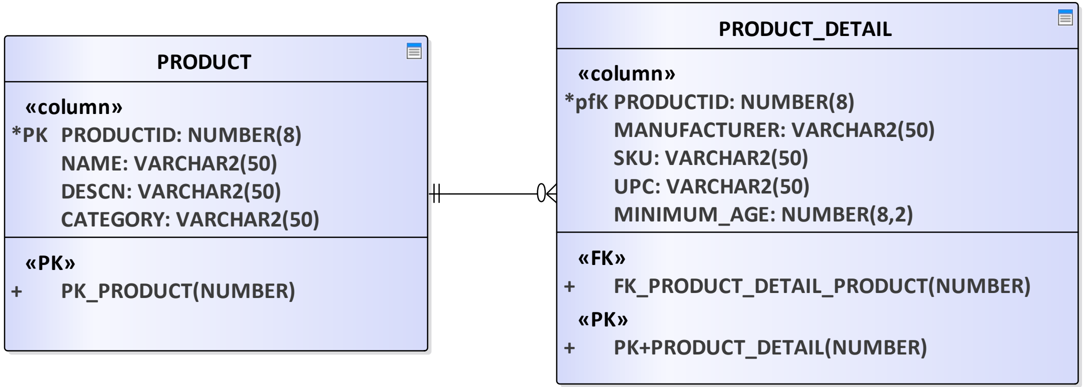
```
<mapper namespace="com.fidelity.integration.ProductMapper">
    <resultMap type="Product" id="ProductMap">
        <id     property="productId"   column="PRODUCTID"/>
        <result property="name"        column="NAME"/>
        <result property="description" column="DESCN"/>
        <result property="categoryId"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

Without `ResultMap`, use column aliases

With `ResultMap`, use raw column names

One-to-One Mapping (DB)

- Database:
  - The PRODUCT_DETAIL table has a productid column that is a foreign key to PRODUCT and is also primary key



PRODUCT

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

«PK»
+ PK_PRODUCT(NUMBER)

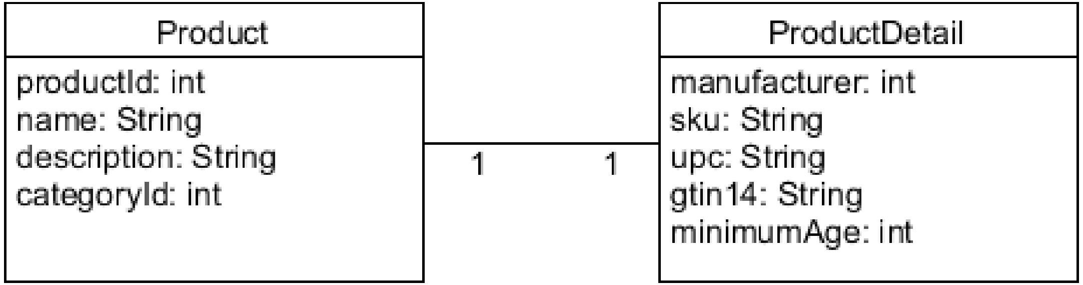
PRODUCT_DETAIL

«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)

- Java:
  - The Product class has a reference to a ProductDetail object



Product

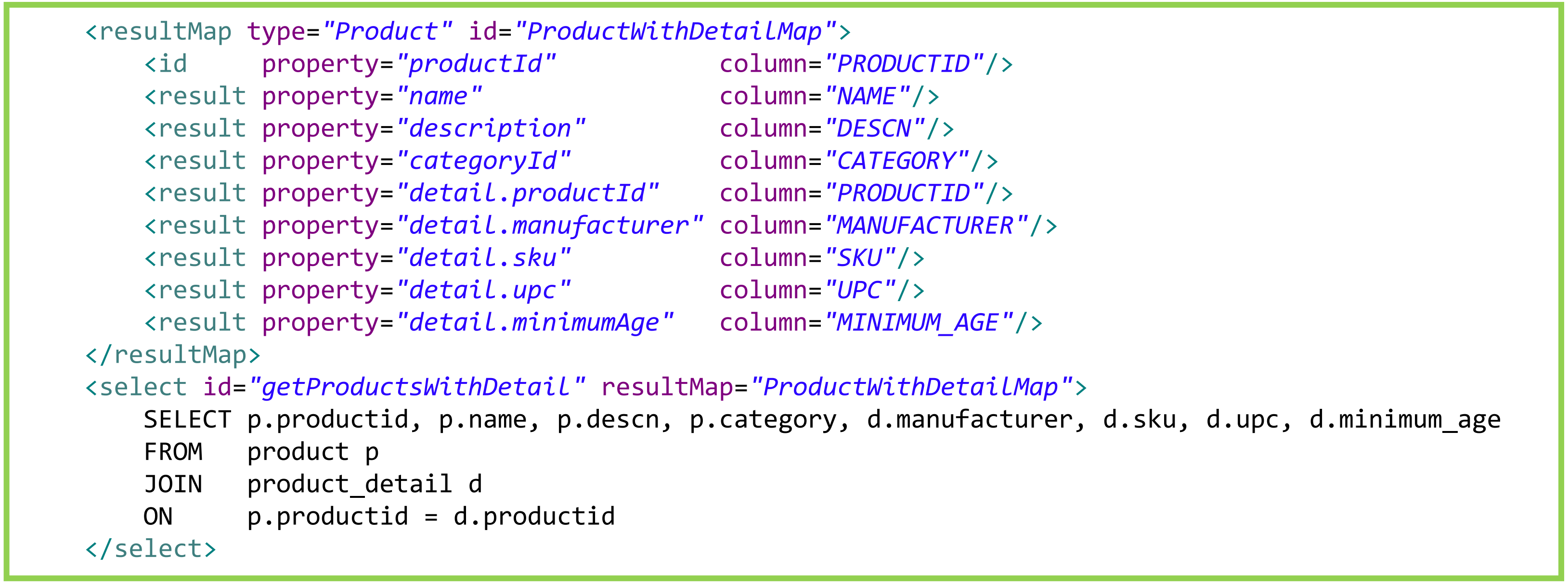
productId: int
name: String
description: String
categoryId: int

ProductDetail

manufacturer: int
sku: String
upc: String
gtin14: String
minimumAge: int

One-to-One Mapping (MyBatis)

- Performing the `getProductsWithDetail` returns fully populated `Product`s



```
<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>
```

- But if we already have the Product mappings defined, we would prefer to reuse them

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 would still have to repeat the mappings
  - We can avoid this by using a Nested `ResultMap` or a Nested `Select` statement

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>
```

Combine Extension and Nesting

- Can combine both techniques to maximize re-use

```
<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>
```

One-to-One Mapping with Nested Select

- Can define an `<association>` element to use another `<select>` statement
  - Set the select attribute to the nested `<select>`
  - Nested select will be executed once for each key value

```
<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

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

<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
  - Again, nested select will be executed once per key value

```
<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 and one-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
- Using advanced MyBatis features
- Configuring MyBatis with Annotations
- Using MyBatis custom type handlers and query caching
- Managing transactions in JUnit tests with Spring

Chapter Concepts

DML Through MyBatis with XML

Transaction Management in Testing

Advanced Topics

SQL Mappers Using Annotations

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 Insert SQL command is configured with an <insert> element

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

Notice the use of property names

- The command is performed by using the Mapper interface object
  – Mapper method ***may*** be defined as returning `int` (the number of rows affected)

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

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

Update

- An Update SQL 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 command is performed by using the Mapper interface object

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

```
@Override
@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 Delete SQL command is configured with a `<delete>` element

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

- The command is performed by using the Mapper interface object

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

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

As before, you may prefer to return the row count

Chapter Concepts

DML Through MyBatis with XML

Transaction Management in Testing

Advanced Topics

SQL Mappers Using Annotations

Chapter Summary

Transaction Manager

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

```
<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

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

Mapper DML methods may be defined as returning `int`, which will be the number of rows affected

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`
    - As described earlier
  - Annotate your JUnit test class with the `@Transactional` annotation
- The TestContext causes all `@Transactional` test methods to rollback automatically
  - The `@Rollback(false)` or `@Commit` annotation can override this default behavior
- The Spring documentation provides more detail on the options that are available
  - https://docs.spring.io/spring/docs/5.0.6.RELEASE/spring-framework-reference/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
    @Rollback(false)
    void testAnotherThing() {
    …
    }
}
```

Annotate class or methods with `@Transactional`

This method will rollback automatically

For some reason, we want this method to commit

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
- Sometimes 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

Helper Function

- Problem: How to find the maximum Department ID?
- Solution: Define and use a helper function to find the maximum Department ID
- Helper functions can be used to find values that are needed before inserting a new record
- Once the maximum Department ID is found, we can add one to use as the id value for a new record

```
private int findMaxDepartmentIdVersion1(List<Department> depts) {
   int maxId = 0;
   for(Department dept : depts) {
     if (dept.getId() > maxId) {
       maxId = dept.getId();
     }
   }
 return maxId;
 }
```

Helper Function Version 2

```
private int findMaxDepartmentIdVersion2(List<Department> depts) {
  OptionalInt maxId = depts.stream()
                          .mapToInt(Department::getId)
                          .max();
  return maxId.orElse(0);
}
```

- Official docs
  - https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
- Help with Streams
  - https://stackify.com/streams-guide-java-8/
  - https://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

Exercise 5.1: DML with MyBatis and Spring

HANDS-ON EXERCISE

60 min

- Follow the directions in your Exercise Manual

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

```
<!-- HyperSQL In-memory Database -->
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>2.4.1</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

Creating a Switchable Installation

- It is often helpful to be able to switch quickly between in-memory and physical databases
  - This is most powerful when done automatically using expressions
    - Out of scope for this course

```
<!-- Define a DataSource for the Oracle 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>

<!-- 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>

<!-- Define an alias for the desired DataSource bean
 Set the name to the DataSource you want to use -->
<alias name="hsqldbDataSource" alias="datasource" />
```

Note that this bean will be instantiated (and the scripts run) even if not used

One datasource is aliased to `datasource` and that name is used everywhere else

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 & 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 5.2: Using an Embedded Database

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Concepts

DML Through MyBatis with XML

Transaction Management in Testing

Advanced Topics

SQL Mappers Using Annotations

Chapter Summary

Autogenerated Primary Keys

- MyBatis can work with primary keys that are automatically generated by the database
  - Use the `useGeneratedKeys` and `keyProperty` attributes in the `<insert>` statement for a database that supports `IDENTITY` columns

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

  - The `keyColumn` attribute is only required for some databases, but is never a problem
    - Where the `IDENTITY` column is not guaranteed to be column 1
    - Specifying it makes `useGeneratedKeys` work with Oracle triggers that contain a sequence (e.g., on older Oracle versions that lack `IDENTITY` support)
  - MyBatis updates the key value of the object passed in

Primary Keys from Sequences

- Older versions of Oracle do not support `IDENTITY` columns
  - If the sequence is used in a trigger, `useGeneratedKeys` works, as shown previously
  - 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

Handling Enumeration Types

- MyBatis supports persisting Java `enum` types
  - An `EnumTypeHandler` is used to handle Java `enum`s
  - The name of the `enum` value is stored in the database

```java
public enum ProductType {
    BIG,
    SMALL
}
```

```java
public class Product {
    private int productId;
…
    private ProductType type;
…
```

```xml
<resultMap type="Product" id="ProductWithTypeNameMap">
    <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_NAME"/>
</resultMap>

<select id="getProductsWithTypeName" resultMap="ProductWithTypeNameMap">
    SELECT productid, name, descn, category, product_type_name
    FROM   product
    WHERE  product_type_name IS NOT NULL
</select>

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

Handling Enumeration Types by Ordinal

- MyBatis supports persisting Java `enum` types by ordinal
  - 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>
```

Custom Type Handler

- Using an `enum` ordinal in the database is not ideal
  - The value may change over time if new values are added
- Can implement a custom type handler
  - Implement `org.apache.ibatis.type.TypeHandler`
  - Or extend `org.apache.ibatis.type.BaseTypeHandler`
- Usually easiest to adapt a standard type handlers from github
- Custom type handlers can be referenced by fully qualified name
  - Or added to the MyBatis configuration
  - `EnumOrdinalTypeHandler` is a standard type handler, but not activated by default

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 with HashMap

- The `<parameterType>` element specifies the type of the input parameter
  - In many cases, can be inferred
- Can use to pass multiple input parameters
  - Put the parameters in a `HashMap`
  - Pass the `HashMap` to the mapped statement
  - `parameterType` is `java.util.Map`

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

```
public interface ProductMapper {
…
    List<Product> getProductsByCategoryAndNameMap(
            Map<Object, Object> map);
…
}
```

```
// Finally, a reason for having the DAO!
@Override
public List<Product> getProductsByCategoryAndNameMap(int categoryId, String name) {
    Map<Object, Object> parameterMap = new HashMap<>(2);
    parameterMap.put("categoryId", categoryId);
    parameterMap.put("name", name + "%");
    return mapper.getProductsByCategoryAndNameMap(parameterMap);
}
```

Passing Multiple Input Parameters by Position

- MyBatis also supports passing multiple parameters to a mapped statement by position
  - Reference the parameters using `#{paramN}` syntax
  - No `parameterType` attribute is needed

```
<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
// to display the third page of 25 records
List<Product> = 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
```

Returning Multiple Results as a Map

- MyBatis supports returning the results of a query as a `HashMap`
  - Specify which property to use for the key
- Here, `products` will contain `productId` as keys and `Product` objects as values

Call SqlSession directly

```
@Override
public Map<Integer, Product> getProductsAsMap() {
    Map<Integer, Product> products = session.selectMap(
            "com.fidelity.integration.ProductMapper.getProducts", // query
            "productId");                                         // key value
    return products;
}
```

```
<mapper namespace="com.fidelity.integration.ProductMapper">
…
    <select id="getProducts" resultType="Product">
        SELECT productId, name, descn as description, category as categoryId
        FROM   product
    </select>
```

Query is unchanged

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

```java
@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

- If the database supports stored procedures, they can be called from MyBatis
- Procedures that do not return results sets are straightforward:
  - To pass more than one parameter, use the parameter map method, or a helper class
  - Can use any of the DML tags (delete, update, insert)

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

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 5.3: 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
  - Invoking the same `<select>` statement with the same SqlSession interface
    - Results are retrieved from the cache rather than the database
- Global second-level caches can be enabled
  - Using the `<cache/>` element in Mapper XML files
  - This can also be customized

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

Exercise 5.4: 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

Advanced Topics

SQL Mappers Using Annotations

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);
…
```

Be careful when concatenating strings across lines

Autogenerated Primary Keys & Sequences

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

```java
@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

```java
@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 `@Results` and `@Result` annotations

```
@Select("SELECT productid, name, descn, category " +
        "FROM   product " +
        "WHERE  productid = #{productId}")
@Results({
    @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);
```

  – 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")
@Results({
    @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="com.fidelity.integration.ProductMapper.getProductDetail"))
})
List<Product> getProductsWithNestedSelect();

@Select("SELECT productid, manufacturer, sku, upc, minimum_age " +
        "FROM   product_detail " +
        "WHERE  productid = #{value}")
@Results({
    @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);
```

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 an `@ResultMap` annotation

```
@Select("SELECT id, name AS cat_name " +
        "FROM   category")
@Results({
    @Result(property="categoryId",  column="ID", id=true),
    @Result(property="name",        column="CAT_NAME"),
    @Result(property="products",    column="ID",
        many=@Many(select="com.fidelity.integration.ProductMapper.getProductListByCategory"))
})
List<Category> getCategoriesWithNestedSelect();
```

Named Parameters

- Annotate parameters with `@Param` to refer to them by name in the query

```
@Select("SELECT productid, name, descn, category " +
        "FROM   product " +
        "WHERE  category = #{categoryId} " +
        "AND    name     LIKE #{name} || '%'")
@ResultMap("com.fidelity.integration.ProductMapper.ProductMap")
List<Product> getProductsByCategoryAndNameParam(
        @Param("categoryId") int categoryId,
        @Param("name") String name);
```

Returning Multiple Results as a Map

- Annotations support Map directly
  - Instead of this:

```
@Override
public Map<Integer, Product> getProductsAsMap() {
    Map<Integer, Product> products = session.selectMap(
            "com.fidelity.integration.ProductMapper.getProducts", // query
            "productId");                                         // key value
    return products;
}
```

  - Use the `@MapKey` annotation

```
@Select("SELECT productid, name, descn as description, category as categoryId " +
        "FROM   product")
@MapKey("productId")
Map<Integer, Product> getProductsAsMap();
```

Optional Exercise 5.5: 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

Advanced Topics

SQL Mappers Using Annotations

Chapter Summary

Chapter Summary

In this chapter, we have explored:

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

Fidelity LEAP

Technology Immersion Program

Mastering Spring and MyBatis

Chapter 6:
Functional Programming

Chapter Overview

In this chapter, we will explore:

- How functional programming is a programming paradigm
  - A style of structuring a computer program
  - Treats computation as the evaluation of mathematical functions
  - Avoids the use of mutable data
- How Java 8 introduced support for functional programming in the Java language
  - Functional Interface
  - Lambda expressions
  - Optional variables
  - Stream API
- How these features make it easier to parallelize Java code

Chapter Concepts

Functional Programming

Lambda Expressions

Stream API

Functional Interfaces

Optional Variables

Chapter Summary

Functional Programming

- Uses expressions (declarations) instead of statements
- The output of a function depends only on the arguments that are passed in
  - Calling the same function twice with the same arguments
  - Produces the same return 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 what is changing that you cannot see

Chapter Concepts

Functional Programming

Lambda Expressions

Stream API

Functional Interfaces

Optional Variables

Chapter Summary

Lambda Expressions

- A lambda expression can be thought of as an anonymous function
  - Can be passed as an argument
  - Has no name
  - Does have a list of parameters, a body, and a return type
- The name lambda comes from the lambda calculus
  - System developed to describe computations
- Provides a more concise way to define and use a callback method

Lambda Expressions (continued)

- Provides a more concise way to define and use a callback method
  - Instead of defining an anonymous inner class

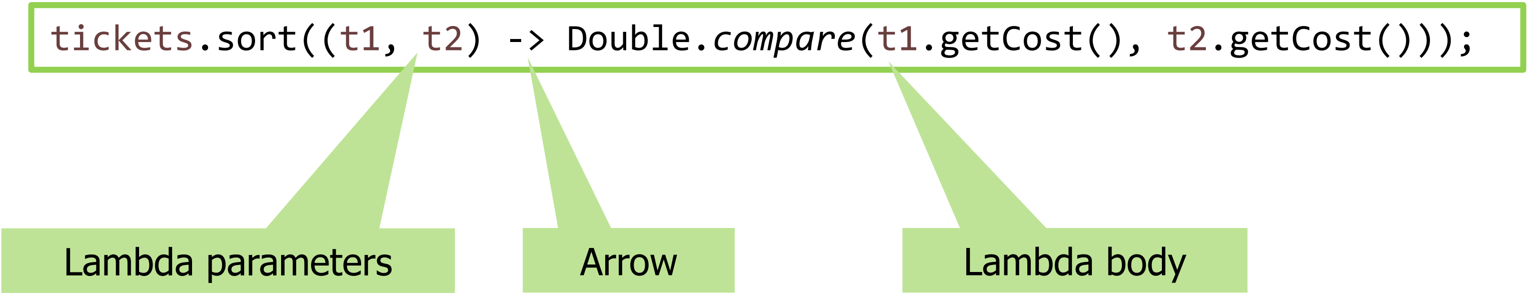
```
tickets.sort(new Comparator<Ticket>() {
    public int compare(Ticket t1, Ticket t2) {
        return Double.compare(t1.getCost(), t2.getCost());
    }
});
```

  - Define the compare method with a lambda expression

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

Lambda Expressions (continued)

- The structure of a lambda expression consists of three sections
  - Lambda parameters
  - Arrow
  - Lambda body



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

Lambda parameters

Arrow

Lambda body

- A lambda expression can be used anywhere an object can
  - It represents a “functional” interface

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

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
- Prefer method references if the lambda just calls another method
- Call a helper function if the body is complex

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

// or this
String::toLowerCase;

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

```
foo(parameter -> helper(parameter));
private String helper(String parameter) {
  // many lines of code
  return result;
}
```

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

Functional Interfaces

Optional Variables

Chapter Summary

Streams

- Streams in Java 8 are designed to:
  - Process collections of values
  - Specify what you want to do
- The implementation manages the scheduling of the operations
  - Multiple threads may be used for different operations
  - The stream library parallelizes the operations
- Streams work on the “what, not how” principle
  - Describe what needs to be done
  - Don’t specify how to carry out the operations

Streams vs. Iteration

- Suppose we have a list of words
  - And we want to find all the 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();
```

- Prefer streams to iterate over a collection
  - More explicit than using for loops and if statements

Stream Characteristics

- Streams do not store their elements
  - The 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 the initial stream into other streams
3. Apply a terminal operation
   a. To produce a result
   b. This will execute any lazy operations
   c. Nothing happens until the terminal operation is called

The stream method creates a stream for the list

The filter method returns another stream. In this case, with words longer than 10 characters.

The count method reduces the stream to a result, a long. 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();
```

Exercise 6.2: Working with Streams

HANDS-ON EXERCISE

20 min

- Follow the directions in your Exercise Manual

Chapter Concepts

Functional Programming

Lambda Expressions

Stream API

Functional Interfaces

Optional Variables

Chapter Summary

What Is a Functional Interface?

- A functional interface:
  - Is an interface that specifies exactly one abstract method
  - The Comparator interface is one example

```
@FunctionalInterface
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

- A lambda expression can provide the implementation
  - Directly inline
  - Treat the whole expression as an instance of a functional interface

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

Using the `@FunctionalInterface`

- Annotate your functional interfaces with `@FunctionalInterface`
  - Compiler will not allow a second abstract method
  - Expresses the intent of the interface

```
@FunctionalInterface
public interface Foo {
   public String fooBar(String str);
}
```

- All interfaces with exactly one abstract method are functional interfaces
  - Regardless of whether they are annotated
  - The annotation just enforces this and protects against future changes

Functional Interfaces Best Practices

- Use default methods sparingly
  - It is easy to add default methods to a functional interface
    - Adding too many default methods is not a good architectural decision
    - Use them as a compromise to upgrade existing interfaces
    - Extending different functional interfaces with the same default method is problematic
- Prefer using standard functional interfaces
  - The `java.util.function` package
    - Defines standard functional interfaces
    - Will meet most needs for lambda expressions and method references
- Do not overload methods with functional interfaces for parameters
  - The compiler will complain about ambiguity

Chapter Concepts

Functional Programming

Lambda Expressions

Stream API

Functional Interfaces

Optional Variables

Chapter Summary

Optional

- The Optional class is a useful tool
  - Helps prevent the dreaded NullPointerException
- Optional wraps around an object
  - With a special marker for empty instead of null

```
Optional<String> optional = Optional.of("Hello");
Optional<String> empty = Optional.empty();
```

- Optional has a method `get()`
  - Returns the value stored in the Optional wrapper
  - But will throw an exception if the Optional is empty
- Use the `orElse(other)` method instead
  - Return the value if present, otherwise return other

```
Optional<Person> personResult = findPerson(name);
Person person = personResult.orElse(Person.GUEST);
```

Using Optional

- Best Practices when using Optional
  - Do not declare an instance variable of type Optional
  - Use null within the private scope of a class
    - Optional is not serializable
  - Use Optional for getters that access an optional field
    - But note that the class is no longer a JavaBean
  - Do not use Optional in setters or constructors
  - Use Optional as a return type
    - For methods that return an optional result

- Optional has three versions for primitive data types
  - `OptionalInt`
  - `OptionalLong`
  - `OptionalDouble`

Chapter Concepts

Functional Programming

Lambda Expressions

Stream API

Functional Interfaces

Optional Variables

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- How functional programming is a programming paradigm
  - A style of structuring a computer program
  - Treats computation as the evaluation of mathematical functions
  - Avoids the use of mutable data
- How Java 8 introduced support for functional programming in the Java language
  - Functional Interface
  - Lambda expressions
  - Optional variables
  - Stream API
- How these features make it easier to parallelize Java code

Key Points

Pattern/Principle	Pointers
Lambda expressions	• Favor expressions over statements • Chain lambda expressions instead of growing them
Functional interfaces	• Define at most one default 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