Mastering the Spring Framework

Chapter 2: Understanding Spring

Chapter Objectives

In this chapter, we will discuss:

- What is a Map?
- Using Spring's dependency injection

Chapter Concepts

Working with Maps

Spring and Dependency Injection

Expression Language

Bean Profiles

External Properties

Chapter Summary

Overview of Collection Framework Interfaces

- `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

- `HashMap` is the most commonly used map
  - `TreeMap` is much less common; it stores data in sort order
  - Each `Entry` in a `Map` is a pair
    - Key
    - Value
- 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

```
// create empty map
Map<Employee, Phone>  directory = new HashMap<Employee,Phone>();

// add items to map (database pseudocode)
while ( dataAccessor.hasMoreRecords() ){
    Employee employee = dataAccessor.getEmployee();
    Phone    phone    = dataAccessor.getPhone();
    directory.put(employee, phone);
}

// search for the Phone for a particular employee
Employee janitor = …;
Phone janitorPhone = directory.get(janitor);
```

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 best performance, the Key class should override `hashCode()` and `equals()`
  - The `hashCode` is usually computed from all the fields of an object
- To avoid the task of writing a `hashCode()` method, some programmers simply use `String` as the Key into a `HashMap`

Chapter Concepts

Working with Maps

Spring and Dependency Injection

Expression Language

Bean Profiles

External Properties

Chapter Summary

Spring Dependency Injection

déjà vu

In Day 1, 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
  - The classes are part of a web application that sells tickets to a museum

Setter Injection ("Wiring")

1. Provide a setter method in the bean

```
public class TicketXMLServiceImpl implements TicketXMLService {
    private ExhibitsDAO exhibitsDAO;
    public void setExhibitsDAO(ExhibitsDAO exhibitsDAO){
         this.exhibitsDAO = exhibitsDAO;
    }
    // rest of class here
}
```

2. *Wire* up the `ExhibitsDAO` in the bean definition for `ticketXMLService`

```
<bean id="dao" class="com. … ExhibitsDAOJDBCImpl"/>

<bean id="ticketXMLService"
      class="com.masteringspring.services.TicketXMLServiceImpl">
          <property name="exhibitsDAO" ref="dao" />
</bean>
```

Call setExhibitsDAO

Pass bean with this id to setExhibitsDAO

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="dao" class="com. … ExhibitsDAOJDBCImpl"/>
<bean id="ticketXMLService"

class="com.masteringspring.services.TicketXMLServiceImpl">
          <property name="exhibitsDAO" ref="dao" />
</bean>
```

Triggers call setExhibitsDAO

- This is the equivalent code:

```
ExhibitsDAO exhibitsDAO= new ExhibitsDAOJDBCImpl();

// any setter methods configured in exhibitsDAO bean definition

TicketXMLService ticketXMLService =
          new com.masteringspring.services.TicketXMLServiceImpl();
ticketXMLService.setExhibitsDAO( exhibitsDAO );
```

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="ticketXMLService"
      class="com.masteringspring.services.TicketXMLServiceImpl">
          <property name="exhibitsDAO" ref="dao" />
</bean>
```

- Using the p-namespace:

```
<beans … xmlns:p="http://www.springframework.org/schema/p"
  <bean id="ticketXMLService"
      class="com.masteringspring.services.TicketXMLServiceImpl"
                                          p:dao="exhibitsDAO" />
  </bean>
```

- The above two configurations are functionally equivalent

Injecting Beans Using Annotations

- Specify that a dependency needs to be injected

```
@Service
public class TicketXMLServiceImpl implements TicketXMLService {
    private ExhibitsDAO exhibitsDAO;
    @Autowired
    public void setExhibitsDAO(ExhibitsDAO exhibitsDAO){
         this.exhibitsDAO = exhibitsDAO;
    }
    // rest of class here
}
```

- `@Autowired` can be on fields, methods, or constructor arguments
- Spring will inject an object of the right type
  - No problem as long as there is only one Spring-managed component that implements `ExhibitsDAO`

@Qualifier

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

```
public class TicketXMLServiceImpl implements TicketXMLService {
          @Autowired
          @Qualifier("dao")
           private ExhibitsDAO exhibitsDAO;
          // etc.
}
```

- It is possible to configure some beans in Java and some in XML
  - Could even have some properties in XML and others with annotations

```
<bean id="dao" … />
<context:annotation-config />
<bean id="ticketXMLService"
       class="com.masteringspring.services.TicketXMLServiceImpl" />
```

Injecting a Value

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

```
public class TicketXMLServiceImpl …{
  public void setMaxAttempts(int maxAttempts){
         this.maxAttempts = maxAttempts;
  }
  public void setUserName(String userName){
         this.userName = userName;
  } …
```

```
<bean id="ticketXMLService"
      class="com.masteringspring.services.TicketXMLServiceImpl">
          <property name="maxAttempts" value="3" />
          <property name="userName" value="jwang"/>
</bean>
```

Injecting a Value with Annotations

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

```
public class TicketXMLServiceImpl {
    private String userName;
    @Value("jChang")
    public void setUserName(String userName){
        this.userName = userName;
    }
}
```

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

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);`**

Invoking the Default Constructor

- When beans are configured like this:
  - The BeanFactory uses the default constructor to create object

```
<bean id="dao"
     class="com.masteringspring.integration.ExhibitsDAOJDBCImpl"/>
<bean id="ticketXMLService"
                 class="com.masteringspring.services.TicketXMLServiceImpl">
      <property name="exhibitsDAO" ref="dao" />
</bean>
```

- The `TicketXMLServiceImpl` and `ExhibitsDAOJDBCImpl` classes must have default (no argument) constructors
  - The field in `TicketXMLServiceImpl` is set by a setter method, not in constructor

Constructor Arguments

- What if a bean has no default constructor?

```
public class TicketXMLServiceImpl implements TicketXMLService {
          private ExhibitsDAO exhibitsDAO;
          public TicketXMLServiceImpl(ExhibitsDAO exhibitsDAO){
             this.exhibitsDAO = exhibitsDAO;
          }
         // more code
}
```

- Need to inject a constructor argument

```
 <bean id="ticketXMLService"
                class="com.masteringspring.services.TicketXMLServiceImpl">
    <constructor-arg ref="dao" />
</bean>
```

Value to pass to single argument constructor

What Can Be Injected Into Constructors?

- The `constructor-arg` element behaves just like the `property` element
  - Can inject bean references, values, list, set, map etc.

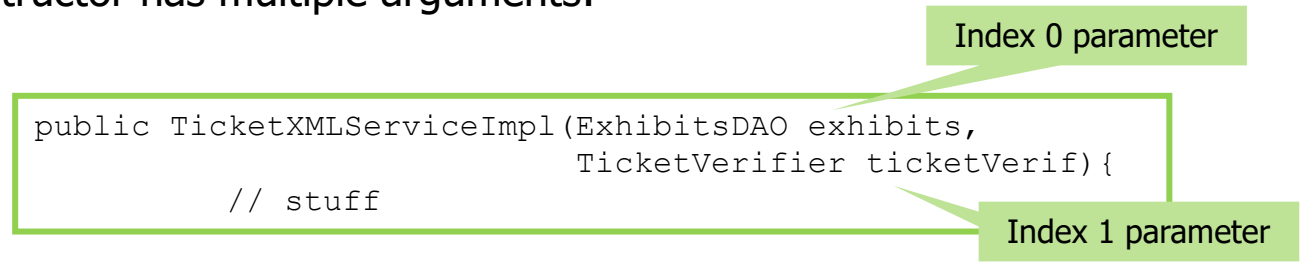
```
<bean id="ticketXMLService"
      class="com.masteringspring.services.TicketXMLServiceImpl">
         <constructor-arg>
            <list>
                 <ref bean="pollockExhibit"/>
                 <ref bean="mooreExhibit"/>
            </list>
         </constructor-arg>
</bean>
```

Exhibit beans to be added to list

```
public TicketXMLServiceImpl(List<Exhibits> exhibits){
         this.exhibits = exhibits;
}
```

Constructor Argument by Index

- If the constructor has multiple arguments:



  - Spring will attempt to match arguments by type
    - If different arguments have same type, then configuration error
  - Safer to explicitly provide index of argument

```
<bean id="ticketXMLService"
              class="com.masteringspring.services.TicketXMLServiceImpl">
   <constructor-arg index="0" ref="dao" />
   <constructor-arg index="1" ref="ticketVerifier" />
</bean>
```

Annotation-Based Construction

- The `@Autowired` annotation supports constructors also

```
@Autowired
public TicketXMLServiceImpl(ExhibitsDAO dao){
          // stuff
```

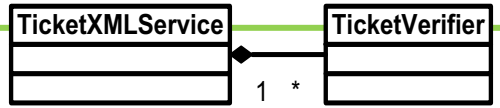
- If you have multiple parameters and are not autowiring by type:
  - Apply `@Qualifier` to the constructor parameters

```
@Autowired
public TicketXMLServiceImpl(@Qualifier("dao") ExhibitsDAO dao,
                            @Qualifier("verifier") TicketVerifier verifier){
          // stuff
```

Fully Configured Beans

- Spring will not inject a bean unless the bean is completely configured
  - `TicketXMLService` depends on a `TicketVerifier` bean

```
<bean id="ticketXMLService"
                  class="com.masteringspring.services.TicketXMLServiceImpl">
   <constructor-arg index="0" ref="ticketVerifier"/>
</bean>
```



  - The `TicketVerifier` bean will have to be completely configured before it is injected as a constructor argument to `TicketXMLService`
    - All its dependencies will have been injected

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);`

Chapter Concepts

Working with Maps

Java Reflection

Spring and Dependency Injection

Expression Language

Bean Profiles

External Properties

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, not a Spring bean

```
<bean id="simulator" class= "com.javamuseum.Montecarlo">
    <property name="seed" value="#{ T(java.lang.Math).random() * 100.0 }"/>
</bean>
```

- Can use properties of Spring beans by chaining them
  - `#{ beanid.property }`

```
<bean id="model" class= "com.javamuseum.Model">
    <property name="startSeed" value="#{ simulator.seed }"/>
</bean>
```

Inject into Model instance result of calling simulator beans seed method

System Properties

- Can inject system properties directly into beans using expression language

```
<bean id="simulator" class= "com.javamuseum.Montecarlo">
    <property name="user" value="#{ systemProperties['java.user'] }"/>
</bean>
```

- This works with annotations also:

```
public class Montecarlo {
   private String user;

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

Chapter Concepts

Working with Maps

Java Reflection

Spring and Dependency Injection

Expression Language

Bean Profiles

External Properties

Chapter Summary

Spring Profiles

- Spring @Profile allow developers to register beans by condition.
  - For example, register beans based on what operating system (Windows, *nix) your application is running, or load a database properties file based on the application running in development, test, staging or production environment.
- Implemented using @Profile annotation

Chapter Concepts

Working with Maps

Java Reflection

Spring and Dependency Injection

Expression Language

Bean Profiles

External Properties

Chapter Summary

External Properties

- Spring @PropertySource annotation is used to provide properties file to Spring Environment. This annotation is used with @Configuration classes.
- Spring PropertySource annotation is repeatable, means you can have multiple PropertySource

Chapter Concepts

Working with Maps

Spring and Dependency Injection

Expression Language

Bean Profiles

External Properties

Chapter Summary

Chapter Summary

- Maps can be used to store values that are retrieved by providing a key
- Spring provides a BeanFactory that supports dependency injection