

Module 5 sample

JXPath and Betwixt: working with XML

| | |
|----------------------------|----|
| 5.1 The JXPath story..... | 1 |
| 5.2 The Betwixt story..... | 21 |
| 5.3 Summary | 34 |
| Index | 35 |

The need to cover two components in this module originated from the consolidation of similar components. JXPath and Betwixt, although different in concept, are complementary technologies that enable a higher degree of interaction between Java objects and XML data. The two technologies, used together, go a long way in simplifying the Java developer's life.

Betwixt resembles Digester, which we covered in the previous module, by providing a similar functionality of converting XML data to Java objects. JXPath is a handy tool if you want shorthand syntax for complex object access.

JXPath and Betwixt solve small but very important pieces of a puzzle. Betwixt helps convert XML data to and from Java objects, whereas JXPath provides an expression language for making sense of complex data structures and objects. Betwixt can convert your data into a readable format, and JXPath can help by accessing this data—even the most complex parts of it.

We'll start with JXPath. It's based on XPath, which is the syntax for traversing XML documents. Therefore, we'll cover the basics of XPath before we dive into JXPath.

We'll explore Betwixt from two angles: the XML-to-Java path and the Java-to-XML path. We'll help you understand how Betwixt operates by first explaining the concept of data binding and then diving into its structure. All through this discussion are plenty of examples to demonstrate how Betwixt operates.

5.1 The JXPath story

JXPath, like the other Commons components, solves a very focused problem for the Java developer. It lets you simplify access to complex nested objects through a well-defined architecture. This architecture derives heavily from XPath, which we'll introduce in section 5.1.2.

Let's start our introduction of JXPath by looking at a real-world problem that we'll use as the basis for discussing this Commons component. As you understand more about JXPath, we'll enhance the application to handle more complex functionality.

5.1.1 Understanding the problem: the Meal Plan application

The whole idea of JXPath is to easily define paths to complex objects. Keeping this in mind, let's create an application that tracks objects, which are hard to access unless you use complex iterations and loops.

The goal is to create a Meal Plan application. The application specifies the meals to be eaten by a person on a diet and tracks the ingredients of each meal. The `MealPlan` class has start and end dates that default to

the current date and seven days from the current date, respectively. You can add meals to the meal plan. Each `Meal` class has an attribute for the day that meal is to be eaten (Sunday = 0, Saturday = 6), the type of meal it is (breakfast = 0, lunch = 1, dinner = 2, other = 3), and the name of the meal. Each meal can contain several ingredients. A key, a name, and an alternate qualify each `Ingredient` class. Figure 5.1 shows a simplistic relationship diagram for these domain objects.

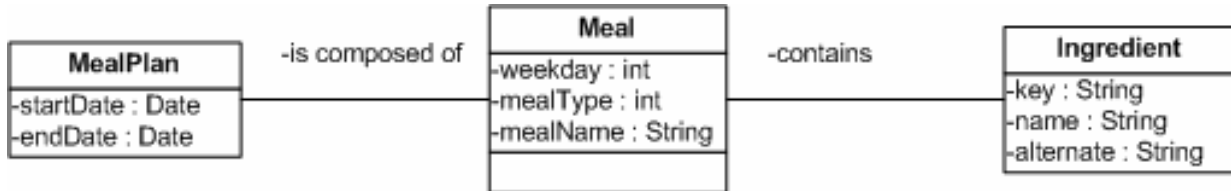


Figure 5.1 Class diagram for the Meal Plan application problem that shows the relationships between domain objects

Let's look at the full source code for these classes. This is our first attempt at creating these objects, and we'll refine them as we explain more about the application and `JXPath`. We'll start with the `MealPlan` class, shown in listing 5.1.

Listing 5.1 The `MealPlan` class

```

package com.manning.common.chapter05;

import java.util.Map;
import java.util.Date;
import java.util.Iterator;
import java.util.Calendar;
import java.util.ArrayList;
import java.util.Collection;
import java.util.GregorianCalendar;

public class MealPlan {
    private Date startDate;
    private Date endDate;
    private Collection meals;    ❶ Declare meals collection

    public Date getStartDate() {
        return this.startDate;
    }
    public void setStartDate(Date startDate) {
        this.startDate = startDate;
    }
    public Date getEndDate() {
        return this.endDate;
    }
    public void setEndDate(Date endDate) {
        this.endDate = endDate;
    }
    public Collection getMeals() {
        return this.meals;
    }
    public void addMeal(Meal meal) {
        if(meal == null)
            throw new IllegalArgumentException("Meal cannot be added null");
    }
}
  
```