

Sample Pages from Capstone Courseware



Various Authors

Sample Pages from Capstone Courseware

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of the author.

A publication of Capstone Courseware, LLC.

877-227-2477

www.capstonecourseware.com

© 1999-2006. All rights reserved.

Table of Contents – Overview

Course Overview

Chapter 1 Our Only Chapter

The instructor often begins a class by previewing the training session using the **course overview**. This is a good time to set expectations, talk about what will and won't be covered, and lay out the class timeline.

Capstone Courseware

- Following are typical pages drawn from a handful of courses.
 - They illustrate the look and feel of a Capstone Courseware book.
 - The [blue captions](#) in the page margins identify the original courses.

Buff boxes like this one point out certain instructional design features, and are not part of the original content.

- As this file is designed to represent typical printed book content, all the text and images are rendered here in black, white, and grayscale.
 - See also the [sample slides PDF](#), which shows the presentation materials, formatted for projection and in full color.
 - On our website you can also get a closer [look at the labs](#), with details on code organization, Eclipse support, JavaDoc, and more.
- Not included here but common to all coursebooks are introductory pages explaining:
 - Course **prerequisites**
 - **Lab software** and file layout

Table of Contents – Detailed

Chapter 1. Our Only Chapter	1
Summarizing Visibility Options.....	3
The Secret Lives of Strings	4
Example: An MVC System	5
Demo: Client-Side Validation	12
Lab 10B: Implementing the Tracking Service	13

Students often find our coursebooks to be valuable desk references after class time. Page headers and footers provide module, chapter, and page numbers to simplify navigation through the book. This **detailed table of contents** also enhances the value of the coursebook book, both during and after class.



CHAPTER 1

OUR ONLY CHAPTER

Courses are organized into chapters, each of which focuses on a specific topic, technology, or skill set. Some courses are also organized into multi-chapter modules. Each module or course follows a logical progression of concepts, and often builds up a practical case study along the way.

Decorations on the chapter title pages vary, to indicate a progression towards a complete understanding of the subject under study.

OBJECTIVES

After completing "Our Only Chapter," you will be able to:

- **See how Capstone courses present technical information in a clear, readable format that**
 - Makes it easy for students to absorb new concepts
 - Supports the instructor's presentation by providing strong visual cues as to the flow of topics
- **Note our use of graphics to highlight the extensive hands-on exercises that are fundamental to every course:**
 - **Examples**
 - **Demos**
 - **Labs**

Each chapter presents its **learning objectives** at the outset. This page is often a convenient basis for an instructor's own overview of the chapter to come. Graphics help to highlight the distinct purposes of different page types; the Objectives page is just one example of this.

The Secret Lives of Strings

- Strings enjoy a special relationship with the compiler.
- The compiler will automatically create a **String** object to wrap a string literal encountered in code.

– That means that this line of code:

```
String x = "Hello";
```

– ... is functionally equivalent to this line of code:

```
String x = new String ("Hello");
```

We've evolved a strong format for our core content pages that allows authors to articulate simple or complex concepts clearly for the reader, while also supporting the instructor's presentation. Bold page titles and emphasized terms make it easy for the instructor to "touch base" with the presentation material without feeling bound too closely to the book text.

- **String** is the only class for which the compiler will do this.
- Note also that **Strings** are **immutable**.
 - For various rather detailed reasons of performance, Java strings implement a **copy-on-write** strategy that allows them to be shared easily without using a lot of memory.
 - The upshot is that once a **String** is created, it cannot change its value.
 - When you think you're changing a string, you're actually creating a new one! So this line of code:

```
x = x + ", Java!";
```

– ... is functionally equivalent to this:

```
x = new StringBuffer (x).append (" , Java!")
  .toString ();
```

HttpServletResponse Methods

Capstone courses are always based on plenty of working code, and instructors can review **code listings** easily, at a nice readable size for projection. Printed code in coursebooks can also be correlated to results of a running example on the screen.

- Having explored the request object, let us get the `HttpServletResponse` object, which implements the `javax.servlet.ServletResponse` interface:

```
public interface javax.servlet.ServletResponse
{
    public void setContentType(String type);
    public void setContentLength(int size);
    public ServletOutputStream getOutputStream();
    public PrintWriter getWriter();
    public void sendRedirect (String URL);
}
```

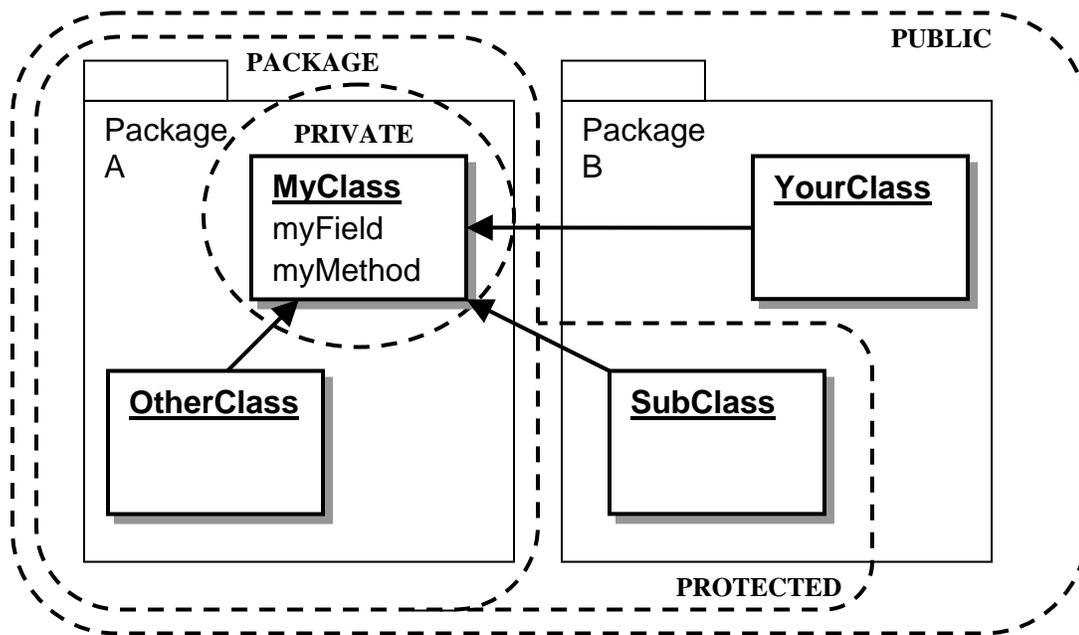
- In our example, we have used two of these methods:

```
response.setContentType ("text/html");  
PrintWriter out = response.getWriter ();
```

- The content type is a MIME type identifier – the most common type will be “text/html”, but “text/plain” and “text/xml” are also common.
- **getWriter** gives us a handy reference to the response output stream as a `java.io.PrintWriter`, which can then be used to produce character output that is handed to the client browser.

Summarizing Visibility Options

- The following diagram illustrates the four visibility options for a class member:



- Only the class itself can see its private members.
- Any class in the package can see members with package visibility.
- Subclasses and package members can see protected members and methods.
- Public members are available to anyone.

We use **diagrams and screenshots** liberally. Nothing else bolsters an instructor's presentation like strong pictorial content. Sometimes a home-grown notation works best, as shown here; but we also find the Unified Modeling Language to be an excellent way to communicate concepts, especially for Java.

- Note, finally, that classes themselves can be either public or package-visible.**
 - Naturally, if one can't see the class, one can't see its members – even if they are public.

An MVC System

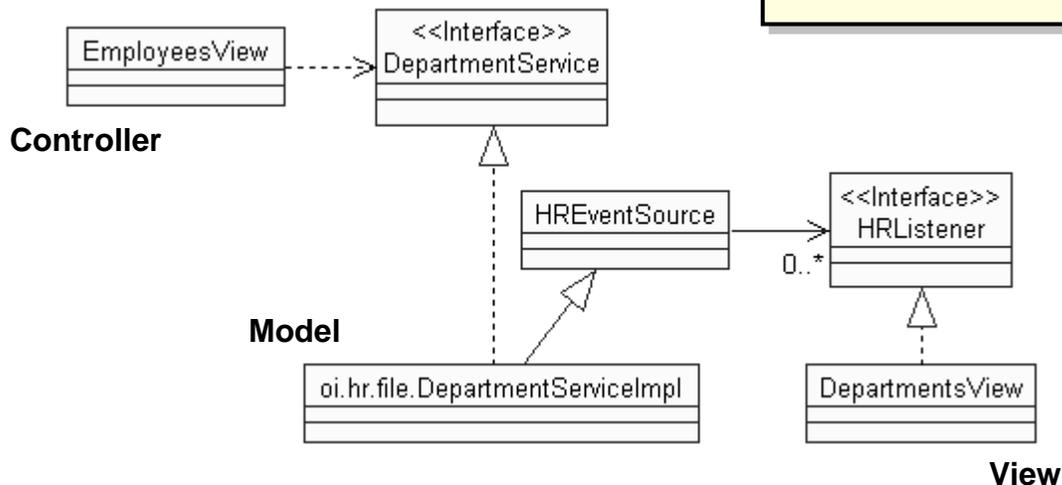
EXAMPLE

- The application in **Examples\HR\Step1** implements an MVC system based on the **HRListener** interface:

```
public interface HRListener
{
    ...
    public void employeeReassigned
        (EmployeeDigest employee,
         DepartmentDigest newDepartment,
         DepartmentDigest oldDepartment);
    public void employeeChanged
        (EmployeeDigest employee);
    ...
}
```

Hands-on coding is essential to Capstone's approach to technical training. Each coursebook comes with a bounty of high-quality code, and students can participate in the learning process by trying out working examples as directed in the book.

Where an even heavier emphasis on hands-on practice is desired, instructors can also direct students to use these working examples as the basis for experimentation with the technology.

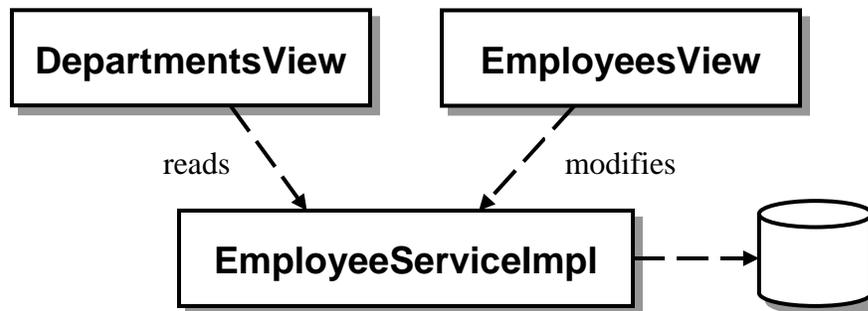


- The model consists of the service implementations, whose mutators fire HR events by calling one or more methods on registered **HRListeners**.
- The three view classes act as **controllers** when they call those mutators, and as **views** by implementing **HRListener** themselves.

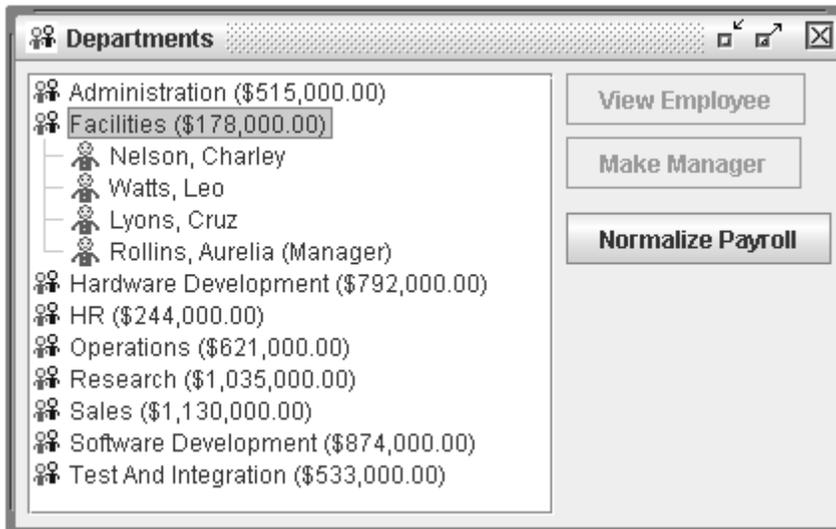
An MVC System

EXAMPLE

- This is how **DepartmentsView** automatically updates its tree when an employee is assigned to a new department from the **EmployeesView**.
 - MVC allows this state information to be centralized in the model, which in turn means that the controller (**EmployeesView** in this case) and view (**DepartmentsView**) needn't have an intimate relationship themselves.



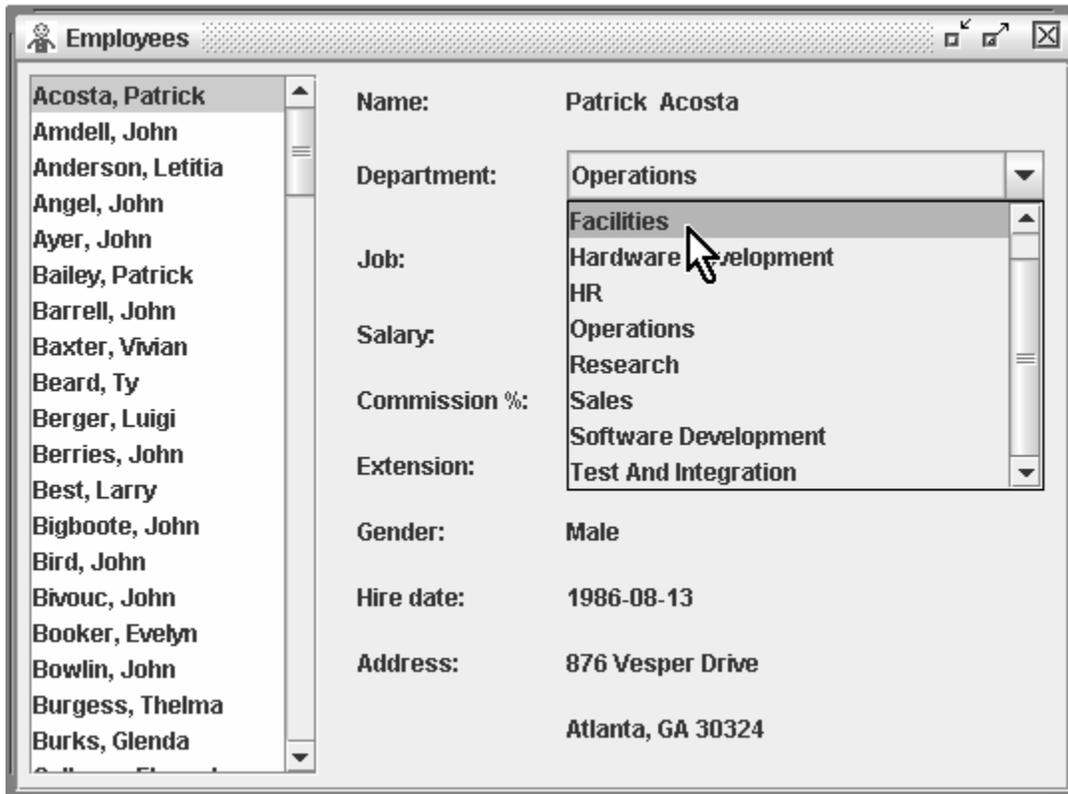
- Try it! **View|Departments** and open Facilities ...



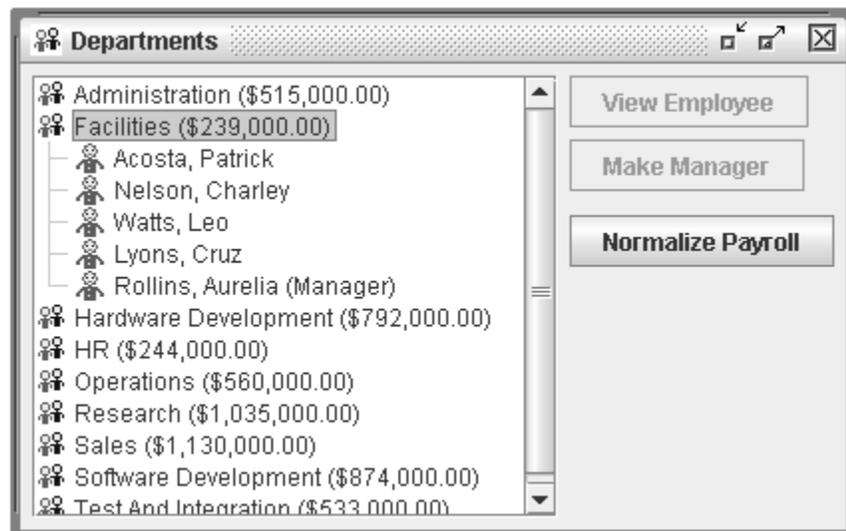
An MVC System

EXAMPLE

- View|Employees, and assign the first employee to Facilities:



- You'll notice the change in the open Departments window:



An MVC System

EXAMPLE

- Let's follow the processing path through source code, to see how this synchronization is achieved.

cc.hr.gui.EmployeesView holds an instance of an inner class **EmployeesPanel** as its only control. This class, in its constructor, connects an event handler to its combo-box of departments:

```
public class EmployeesView
->public static class EmployeesPanel
->public EmployeesPanel ()
{
    cbDepartment.addItemListener
        (new DepartmentHandler ());
}
```

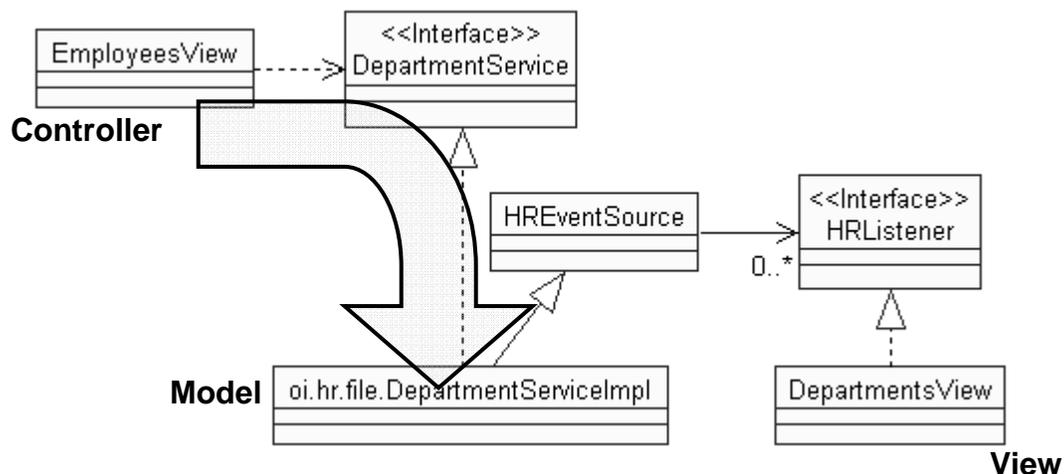
An MVC System

EXAMPLE

Thus when the user chooses a new department for a selected employee in the Employees window, the method **itemStateChanged** in **DepartmentHandler** (itself an inner class of **EmployeesPanel**) does a bit of error checking, and then calls **DepartmentService.assignEmployee** to make the reassignment.

```
public class EmployeesView
->public static class EmployeesPanel
  ->protected class DepartmentHandler
    ->public void itemStateChanged (ItemEvent ev)
    {
      dService.assignEmployee
      (selectedDept, selectedEmployee);
    }
}
```

This then is the **controller** writing to the **model** – though the controller in this system is also an observer on a JFC control.



- The warning-sign behavior here would have been an immediate call directly to the **DepartmentsView**, telling it to refresh itself.

An MVC System

EXAMPLE

The service implementation in play, which is `cc.hr.file.DepartmentServiceImpl`, carries out the changes to the model – but it also **fires a model event**:

```
public class DepartmentServiceImpl
->public void assignEmployee (...)
{
    ...
    pDept.addEmployee (pEmp);

    for (HRListener recipient : getListeners ())
        recipient.employeeReassigned
            (pEmp, pDept, oldDepartment);
}
```

Who are the listeners for model events on the department service? Well, if there is a department window open at the moment, it will be one of those listeners: it takes a reference to the department service in its constructor, and hooks its own lifecycle methods to assure that it is attached as an observer for any model events. See `cc.hr.gui.DepartmentsView`:

```
public class DepartmentsView
->public DepartmentsView (...)
{
    addInternalFrameListener
        (new WindowHandler (source));
}

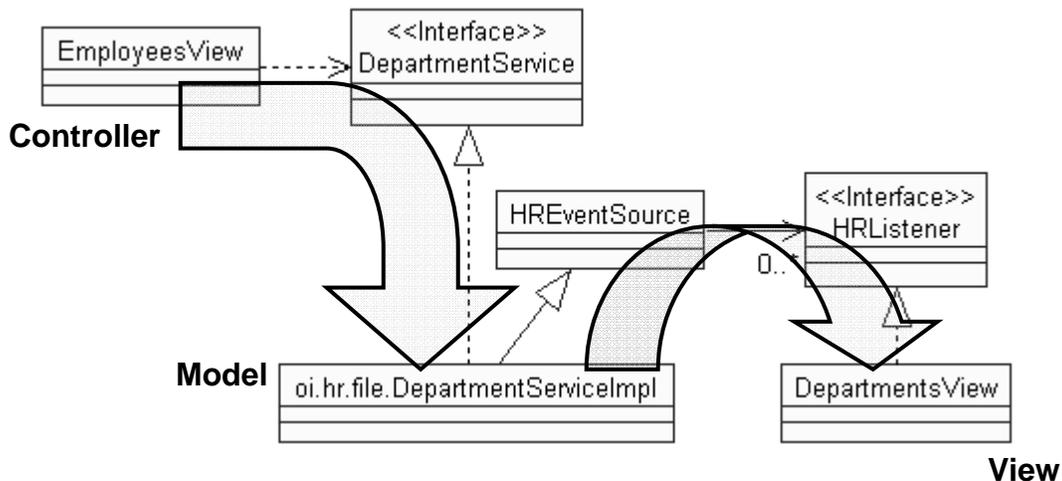
public class DepartmentsView
->protected class WindowHandler
->public @Override void internalFrameOpened (...)
{
    source.addHRListener (mainPanel);
}
```

An MVC System

EXAMPLE

So when the model fires an event, this view is listening! and immediately updates its display as necessary to reflect the change:

```
public class DepartmentsView
->public static class DepartmentsPanel
  ->public void employeeReassigned (...)
    // Removes the employee as a tree node from
    // his or her old department and adds to the
    // new department ...
```



- Thus the departments view stays in sync with a change made in the employees view.
- The system is centralized such that all changes flow “down” to the model and back “up” to views that care about certain types of changes.
- However, the system is not complete!
 - In the upcoming lab you’ll observe a warning sign for MVC that indicates that one of the views has not yet subscribed for model events, and so is missing out on real-time changes.

Client-Side Validation

DEMO

- We'll add client-side validation to the Ellipsoid application.
- Work in **Demos\JavaScript\Step1**.
 - The completed code is in **Step2**.

1. Open **View/index.jsp** and change the `<form>` `<html:form>`. Also, add the `onsubmit` HTML validation method:

```
<html:form action="/Compute.do"
  onsubmit="return validateEllipsoid()" >
  <center><table border="0" cellpadding="8" >
```

Going beyond a static example, a Capstone demo provides step-by-step instructions that allow students to follow along with the instructor as he or she carries out a simple development task, such as adding validation logic to an existing HTML form.

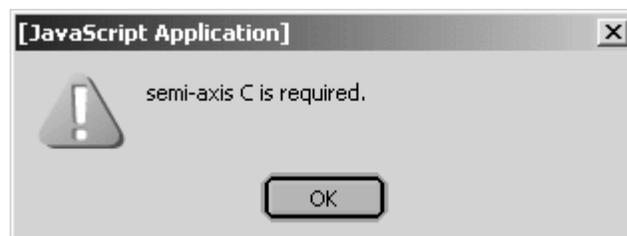
Most often, students are best served by a progression from observing (Examples) to supervised work (Demos) to solo work and experimentation (Labs).

2. Add the `<html:javascript>` tag as the last element in the HTML body:

```
<html:javascript formName="Ellipsoid" />
```

```
</body>
```

3. Rebuild the application and try providing bad values:



4. You may want to take a look at the HTML source as delivered by the `<html:javascript>` tag – pretty good ratio!

Implementing the Tracking Service

LAB 10B

In this lab you will complete the implementation of the Order-Tracking service that you used in Lab 10B. Now the source WSDL and completed JAX-RPC client are provided, and we travel back in time to work through the implementation of the service side.

Detailed instructions are contained in the Lab 10B write-up at the end of the chapter.

Suggested time: 30 minutes.

Most commonly a chapter will ease students into more complex hands-on exercises: first with prepared examples, then with instructor-led demonstration, and finally by giving the students their own lab time to tackle a new development problem on their own.

Lab "announcement" pages such as this one indicate the appropriate stopping points for lab work; students then refer to detailed instructions at the end of the chapter.

SUMMARY

Each chapter concludes with a **summary** of lessons learned. The Summary page pairs up nicely with the Objectives at the beginning of the chapter and gives the instructor a good opportunity to review and also to solicit additional questions from students and to reinforce the concepts presented so far.

- **Hopefully this document gives you a strong sense of the instructional design of Capstone Courseware.**
- **We believe that our attention to the details of visual presentation helps both instructors and students to have a rewarding teaching and learning experience.**
- **If you have questions, or would like to learn more about Capstone courseware, please ...**
 - Visit our website:
<http://www.capstonecourseware.com>
 - Contact us – we'll be only to happy to answer questions or assist you with finding the right course materials for an upcoming training:
877-227-2477
info@capstonecourseware.com
- **Note that a sample of our detailed lab instructions follows this page.**
 - Detailed instructions are produced only in the printed book; they are not intended for instructor presentation and so don't appear in the accompanying sample slides.

Web-Service Namespaces

- Following is a list of the exact URLs to use for the known namespaces for SOAP and WSDL

After delving into a given API or language, students often want a simple reminder of basic usage: keywords, identifiers, basic grammar, and common code phrases. Many of our chapters and whole coursebooks include **quick-reference** appendices that serve this purpose.

Name	Common Prefix	URL
XML Schema	xs: or xsd:	http://www.w3.org/2001/XMLSchema
Instance	xsi:	http://www.w3.org/2001/XMLSchema-instance
SOAP		
Envelope	soap-env:	http://schemas.xmlsoap.org/soap/envelope/
Encoding	soap-enc:	http://schemas.xmlsoap.org/soap/encoding/
WSDL		
WSDL	wSDL:	http://schemas.xmlsoap.org/wSDL/
SOAP Binding	soap:	http://schemas.xmlsoap.org/wSDL/soap/
HTTP Binding	http:	http://schemas.xmlsoap.org/wSDL/http/
UDDI		
UDDI Schema	uddi:	urn:uddi-org:api_v2
Pub. WSDL		urn:uddi-org:publication_v2
Inquiry WSDL		urn:uddi-org:inquiry_v2

Implementing the Tracking Service

LAB 10B

Introduction

In this lab you will complete the implementation of the Order-Tracking service that you used in Lab 10B. Now the source WSDL and completed JAX-RPC client are provided, and we travel back in time to work through the implementation of the service side.

We'll take a similar approach to Lab 10A here, in that we'll implement the service endpoint implementation first, and then to go from WSDL, and to see how close you come to the target. You will adjust to get a clean compile, deploy, and test.

Suggested Time: 30 minutes.

Root Directory: c:\Capstone\WSJava

Directories:

Labs\Lab10B	(do your work here)
Examples\Tracking\Step1\Server	(backup copy of starter files)
Examples\Tracking\Step2\Server	(contains lab solution)
Examples\Tracking\Step2\Client	(to test completed service)

Files:

- cc\biz\TrackingImpl.java
- cc\biz\OrderDB.java
- WSDL\OrderTracking.java

Packages: cc.biz

Instructions

1. Open the **OrderTracking.wsdl** descriptor and review the contents.
2. Create a new source file **TrackingImpl.java** in a new **cc\biz** directory. Create a skeletal implementation for the **Tracking** port type in this class. Set the package to **cc.biz**, and write out the necessary method signatures, based on the information in the WSDL file. The goal for this step is an implementation file that lists all the methods and returns hard-coded values, just so that it will compile cleanly.

A major advantage of Capstone courseware rests in its **detailed lab instructions**. Many providers offer only "sink or swim" labs with one-paragraph problem definitions. We find that students benefit much more from step-by-step instructions that carry them through their first experience with new technology.

For more on Capstone's lab software, see **Look Inside the Labs**.

3. Run **asant** to generate Java from the WSDL file and to compile your implementation class. If you get a clean compile here, skip to the next step. If not, double check your code, compare it to the generated **Tracking_Impl** in **Build\cc\biz**, and/or work through the following checklist.
 - The class must implement the **Tracking** interface.
 - The **checkStatus** method must take a **long** and return a **StatusEnum**.
 - The **getDetail** method must take a **long** and return an **Order**.
 - The **cancel** method's signature is identical to that of **checkStatus**.
 - All methods must be public.
 - All methods must declare that they throw the **java.rmi.RemoteException**.
 - All methods must return some value: **null** will do for now.
4. Once your class is compiling cleanly, you can implement the actual service logic. The hypothetical order database is already prepared in a separate class **OrderDB**. Add a private field to **TrackingImpl** of this type, called **DB**, and initialize it to a new **OrderDB** instance.
5. Implement **checkStatus** by calling **DB.getOrder**, passing the **ID** argument – this returns an **Order** instance – and getting and returning its status value.
6. **getDetail** is one step simpler: you can just return the results of **DB.getOrder**.
7. Implement **cancel** by getting the order object into a local called **order**. Now set the status to the desired value – thanks to an uninspired mapping of enumerated values to constant identifiers this is **StatusEnum.value6**. Then return the status from the object, just as you do in **checkStatus**.
8. Rebuild the project and fix any compile errors. From **Examples\Tracking\Step2\Client**, build and run the client application, and confirm output as shown below.

```
asant
Track check 2
Back-ordered
Track detail 2
Order details:
  ID:          2
  Customer ID: 3456
  Status:      Back-ordered
              100 Darjeeling
              25 Lapsang Souchong
```

9. If you are not getting the correct behavior, check over code again, consult your instructor, or possibly try redirecting the client through the SOAPSniffer tool to inspect the SOAP traffic. You can also send the prepared message in **SOAP\TestInvocation.txt** to the service using SOAPPad.