COMP/EECE 7012

Exam 1

Spring 2014

Name (Last, First): Solutions

Rules:

- No potty breaks.
- Turn off cell phones/devices.
- Closed book, closed note, closed neighbor.

- WEIRD! Do not write on the backs of pages. If you need more pages, ask me for some.

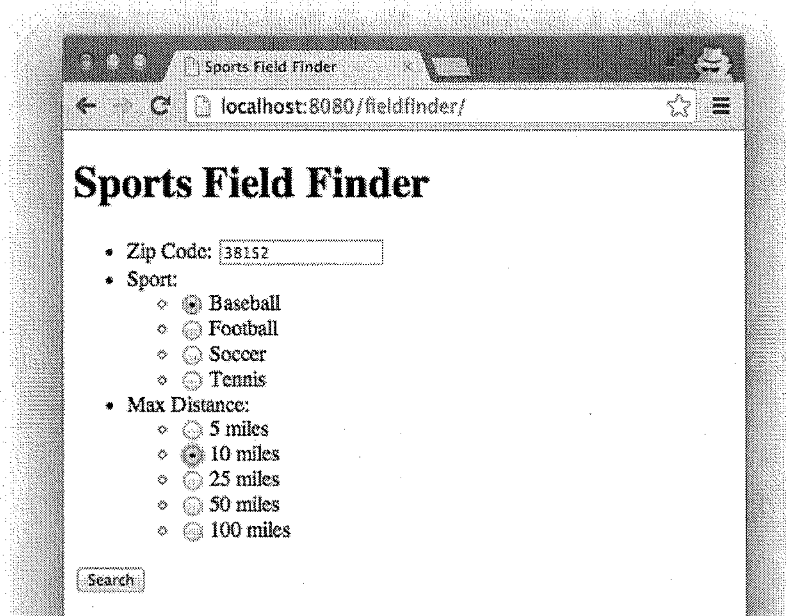
Reminders:

- Verify that you have all pages.
- Don't forget to write your name.
- Read each question carefully.
- Don't forget to answer every question.

Additional Items:

- For questions that involve writing code:
  - You may omit `import` statements.
  - You may omit exception-handling code.

Imagine an web application for finding sports fields in your area. The app might have a search page into which the user can enter their criteria, like this:



Then, the app might return search results that list the names of parks that meet the criteria, like this:

In the search results, the dollar signs appear for parks that are not free.

For the next two questions, your job will be to reverse engineer portions of such a web app. Use the following figures to aid in your solution—read them thoroughly!

- Figure 1: Explains what servlets/JSPs there are, and how they should fit together.
- Figure 2: Shows the HTML code for the above form.
- Figure 3: Shows some "plain old" Java classes that you might use in your solution.
- Figure 4: Summarizes some relevant Java API classes/interfaces.

It is vitally important that all your solutions follow the <u>MVC architectural pattern</u> that we learned in class.

1. [10pts] Write a <u>complete</u> class that implements the **FindFieldsServlet** servlet. You may omit code that validates input values, that handles error cases, and that has to do with serialization.

```
@WebServlet("/findFields.do")
public class FindFieldsServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws ServletException, IOException {
        String zipCode = request.getParameter("zipCode");
        String sport = request.getParameter("sport");
        String maxDistance = request.getParameter("maxDistance");
        FieldFinder fieldFinder = new FieldFinder();

        List<FieldLocation> fields =
                fieldFinder.findFields(zipCode, sport, maxDistance);
        request.setAttribute("fields", fields);

        String viewPath = "/WEB-INF/searchResults.jsp";
        RequestDispatcher view = request.getRequestDispatcher(viewPath);
        view.forward(request, response);
    }

}
```

2. [10pts] Write a <u>complete</u> JSP that implements **searchResults.jsp**. Your solution should contain the following HTML elements: body, html, head, h1, li, title, ul, !DOCTYPE. Here are the first 2 lines:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ page import="all-the-imports-go-here"%>
```

```
<%
    List<FieldLocation> fields =
        (List<FieldLocation>) request.getAttribute("fields");
%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Search Results</title>
</head>
<body>

    <h1>Search Results</h1>

    <ul>
        <%
            for (FieldLocation field : fields) {
        %>

        <li><%=field.getName()%>
            <% if (!field.isFree()) { %>($$$)<% } %>
        </li>

        <%
            }
        %>
    </ul>

</body>
</html>
```

Given the Sports Field Finder database tables in Figure 5, give the results of the following queries. Don't forget to label the columns. Leave any unused cells empty.

3. [4pts] `SELECT * FROM `park` WHERE `name` LIKE 'Kicky Hills';`

id	name	sport	freePark
444	Kicky Hills	baseball	0

4. [4pts] `SELECT * FROM `park` WHERE `freePark` = '1';`

id	name	sport	freePark
222	Sporty Hills	baseball	1
333	Runny Hills	baseball	1

5. [8pts]

```
SELECT `name`,`freePark` FROM `park` INNER JOIN `parkDistance`
    ON `park`.`id` = `parkDistance`.`parkId`
    WHERE `zip` = '38152' AND `distance` <= '5.0';
```

name	freePark
Sporty Hills	1
Runny Hills	1

6. [10pts] Consider the following scenario involving Subversion (SVN). To help jog your memory of SVN commands, here is a list of some SVN commands with some other nonsense mixed in:
Insert, Import, Commit, Pull, Checkout, Update, Upgrade, Branch, Merge, Put.

Scenario: You want to contribute to the Sports Field Finder project, so you contact the project lead, Alice. With respect to SVN, what info must Alice give you so that you can get started with making contributions?

(1) Repository URL
(2) Login/password

Once you have that info, which SVN command should you run to get a working copy?

Checkout

Once you get a working copy of the code, you start making changes. Eventually, you're ready to push your changes into the repository. What SVN command should you run?

Commit

When you run the command, SVN reports an "out of date" error. What must have happened that would cause this error?

For at least one of the files you modified, another person committed changes to the file in the time since you checked it out. Thus, your copy of the file is out of date with respect to the version in the repository.

Given the error, what is the next SVN command that you should run?

Update

7. [10pts] Create a complete JUnit test case (including the surrounding class) that tests the **findFields()** method of class **FieldFinder**. Specifically, your test should use these inputs: "38152", "football", and "5". The expected output with these inputs should be an empty list (i.e., there are no football fields that meet this criteria). Hint: The only JUnit stuff you need is **@Test** and **fail()**.

```
public class FieldFinderTest {

    @Test
    public void testFindFields() {
        String inputZip = "38152";
        String inputSport = "football";
        String inputMaxDistance = "5";
        FieldFinder finder = new FieldFinder();
        List<FieldLocation> fields =
                finder.findFields(inputZip, inputSport,
                        inputMaxDistance);
        if (fields == null || !fields.isEmpty()) {
            fail();
        }
    }

}
```

For each of the following objects, assuming your application needs to call the object's **getAttribute()** and **setAttribute()** methods, must a servlet that uses the object synchronize accesses of the object? Explain why or why not (and be more detailed than just "multiple threads can/cannot access the object").

8. [2pts] The **request** object (of type **HttpServletRequest**)?

No. There is only 1 thread and 1 request object per HTTP request. Thus, request is "thread local" (not accessed by any other threads).

9. [2pts] The **session** object (of type **HttpSession**)?

Yes. There is one session object per user session. However, a user session may involve multiple concurrent HTTP request, and thus, multiple concurrent threads.

10. [2pts] The servlet context (aka **application**) object (of type **ServletContext**)?

Yes. The servlet context is shared by all requests, and thus, by multiple threads.

11. [3pts] For the following classes/JSPs, what parts of the MVC pattern do they correspond to (M, V, or C)? For full credit, you must spell out the word, not just give the abbreviated initial.

Class FieldFinder Model

JSP searchResults.jsp View

Class FindFieldsServlet Controller

Programming Skills Question

12. [10pts] Write a complete method **freeCount()** in the class below such that it takes as its lone parameter a list of sports fields (same type as returned from **FieldFinder.findFields()**), and returns an integer count of the number of fields in the list that are free. (Note: The method should not print anything.)

```
public class FieldListPrinter {

    public int freeCount(List<FieldLocation> fields) {
        int result = 0;

        for (FieldLocation field : fields) {
            if (field.isFree()) {
                ++result;
            }
        }

        return result;
    }

}
```

Figures for the Sports Field Finder Web App

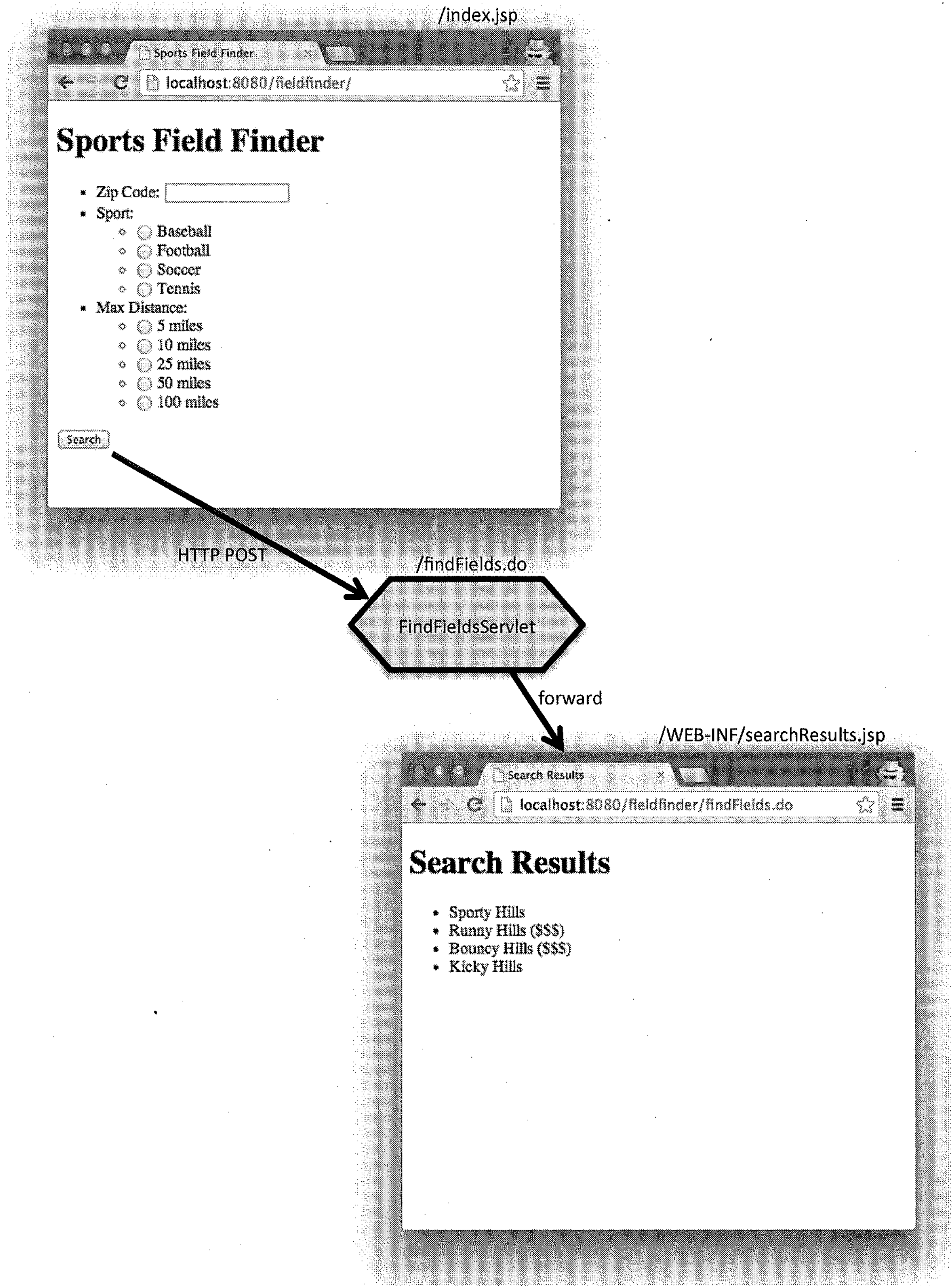


Figure 1. Servlet/JSP diagram for the Field Finder web app.

```
<form method="post" action="<%=application.getContextPath() %>/findFields.do">

	<ul>
		<li>Zip Code: <input type="text" name="zipCode">
		</li>

		<li>Sport:
			<ul>
				<li><input type="radio" name="sport" value="baseball">
					Baseball</li>
				<li><input type="radio" name="sport" value="football">
					Football</li>
				<li><input type="radio" name="sport" value="soccer">
					Soccer</li>
				<li><input type="radio" name="sport" value="tennis">
					Tennis</li>
			</ul>
		</li>
		<li>Max Distance:
			<ul>
				<li><input type="radio" name="maxDistance" value="5">
					5 miles</li>
				<li><input type="radio" name="maxDistance" value="10">
					10 miles</li>
				<li><input type="radio" name="maxDistance" value="25">
					25 miles</li>
				<li><input type="radio" name="maxDistance" value="50">
					50 miles</li>
				<li><input type="radio" name="maxDistance" value="100">
					100 miles</li>
			</ul>
		</li>
	</ul>
	<p>
		<input type="submit" value="Search">
	</p>

</form>
```

Figure 2. HTML code for the form in index.jsp.

```java
/**
 * JavaBean representing a sports-field location.
 */
public class FieldLocation {

	private int id = 0;
	private String name = "";
	private String sport = "";
	private boolean freePark = true;

	public FieldLocation() {}

	public FieldLocation(int id, String name, String sport, boolean freePark) {
		this.setId(id);
		this.setName(name);
		this.setSport(sport);
		this.setFree(freePark);
	}

	public int getId() { return id; }
	public String getName() { return name; }
	public String getSport() { return sport; }
	public boolean isFree() { return freePark; }

	public void setId(int id) { this.id = id; }
	public void setName(String name) { this.name = name; }
	public void setSport(String sport) { this.sport = sport; }
	public void setFree(boolean freePark) { this.freePark = freePark; }
}
```

```java
/**
 * Responsible for finding sports fields based on various criteria.
 */
public class FieldFinder {

	/**
	 * Searches for sports fields that meet the specified criteria. This method is
	 * thread safe, and as such, may be called without any synchronization.
	 * @param zip The zip code of the starting point.
	 * @param sport The type of sports field to search for.
	 * @param maxDistance Do not return fields farther than this distance from the
	 *     starting zip.
	 * @return A list of sports fields that meet the search criteria.
	 */
	public List<FieldLocation> findFields(String zip, String sport,
	   String maxDistance) {
		...
	}
}
```

Figure 3. Two given classes from the Field Finder code base. You need not implement these classes, just use them.

Class **HttpServlet**

- Annotation to declare URL pattern: **@WebServlet**(*url-pattern-string-goes-here*)
- protected void **doGet**(HttpServletRequest req, HttpServletResponse resp)
- protected void **doPost**(HttpServletRequest req, HttpServletResponse resp)

Interface **HttpServletRequest**

- String **getParameter**(String name)
- Object **getAttribute**(String name)
- void **setAttribute**(String name, Object o)
- RequestDispatcher **getRequestDispatcher**(String path)

Interface **RequestDispatcher**

- void **forward**(ServletRequest request, ServletResponse response)

Interface **List**<E>

- boolean **add**(E e)
- E **get**(int index)
- int **size**()

Figure 4. Excerpts from the Java API.

park

id	name	sport	freePark
222	Sporty Hills	baseball	1
333	Runny Hills	baseball	1
444	Kicky Hills	baseball	0

parkDistance

zip	parkId	distance
38152	222	2.300
38152	444	6.400
38152	333	1.200
90210	222	1812.300
90210	444	1867.400
90210	333	1806.200

Figure 5. Two database tables used by the Sports Field Finder web app. In table `park`, `id` is a primary key, and in table `parkDistance`, `parkId` is a corresponding foriegn key.