Practical Ajax Projects with Java™ Technology

Frank W. Zammetti

Apress®

Practical Ajax Projects with Java™ Technology

ISBN-13 (pbk): 987-1-59059-695-1

ISBN-10 (pbk): 1-59059-695-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Chris Mills
Technical Reviewer: Herman van Rosmalen
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editor: Liz Welch
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Gunther
Compositor: Lynn L'Heureux
Proofreader: Linda Seifert
Indexer: Brenda Miller
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 10

AJAX Warrior: Back to the Future in a Fun Way!

Well, the road has been long! We've explored six other applications together, learned a great deal about using Ajax in various ways, and produced some really useful stuff from it all. Now we stand on the verge of a great journey. You are about to embark on a quest of understanding, and of entertainment. We will control the vertical. We will control the horizontal. We will venture forth into a land of strange creatures, magic, powerful enemies, and heroic deeds. As your guide, I'll take you into our final Ajax project, and what a dandy it is: a game! In this chapter you'll see more ways of using "naked" Ajax. You'll learn about something called JSON, a data interchange format that is all the rage in these Web 2.0 days, and you'll see a good amount of CSS and DOM scripting techniques. You'll be introduced to even more JavaScript tricks and even produce another kind of server-side application framework. At the end of it all, you get to slack off a bit and play a game that I hope you'll find entertaining as well as educational. Let the adventure begin!

Requirements and Goals

AJAX Warrior tells the story of the mythical land of Xandor. The evil Lord Mallizant has stolen the five sacred artifacts of Xandor: the Crystal Skull, the Scroll of Life, the Medallion of the Sun, the Staff of Tiuwahha, and the Ankh. He has banished the good king Chimley from his home in Castle Faldon and hidden the artifacts throughout the lands, protected in stone and guarded by evil magic. However, before Chimley was banished, he managed to steal the five keys needed to open the magical doors, and he has scattered three of them throughout the communities of Xandor and given two of them to custodians.

Your job, as the AJAX Warrior, is to first retrieve the five keys, and then retrieve the five artifacts. Once all five artifacts are in your possession, they will be reunited, and Mallizant will instantly die (he is now tied to their fate, much like Sauron in *Lord of the Rings*). Along the way you'll have to do battle with Mallizant's minions, a variety of beasts with varying degrees of fighting abilities. You'll need to speak to the inhabitants of Xandor because remember, two of them will have keys you need, and some will give you clues on where to find the other keys and even the artifacts themselves. Oh, and be sure not to kill one of the key masters; needless to say, your quest will be unceremoniously cut short if you do that!

OK, back here in the real world…

What exactly are we going to accomplish with this game? What are some of the specifics we're going to implement? Let's now enumerate at least a few of them:

- We'll have maps that are 100×100 tiles in size, where each tile is 32×32 pixels in size. This should give us a decent-sized world to inhabit.
- There will be four communities in Xandor: Castle Faldon, an unnamed village, and two towns named Rallador and Triyut. Each of them is also represented by a 100×100 map.
- The player will be able to cast a number of different spells, including a healing spell, a Fire Rain spell (for combat), and a Freeze Time spell (to stop the passage of time for every being but the player).
- The player will be able to possess a number of weapons, including a dagger, staff, and mace.
- We should be able to save a game in progress, and continue that game at will. Of course, that save should be done on the server.
- We'll be able to talk to some characters in the game. Some characters will be belligerent, though, and will not talk to us, but will instead attack us all the time.
- When talking to a character, we'll use a script system that allows for some variability depending on what the player says. As we reply to the character, their "karma" will increase or decrease depending on our responses. If their karma reaches zero, they will run away. If we're talking to a key master, we have to get their karma to 15, at which point they'll give us the key. So, it is important for the player to talk "properly" to each character.
- The player will be able to view their inventory of spells and weapons, will be able to call up help at any time, and will be able to cast spells at any time. They'll also be able to toggle between Attack mode, which means they'll attack any character they encounter, and Talk mode, which means they'll enter into conversation with any nonbelligerent character they encounter (provided they speak the language of the character!).
- Lastly, we want to have the vast majority of the true game logic on the server, *not* the client. The client should, for the most part, just be responsible for showing outcomes. We'll also do all our Ajax functions "nakedly," that is, without using any Ajax library, and we'll also use a new communication technique: JSON.

Whew, that sure is a lot of work! And just to remind you, these are only *some* of the goals. We'll discover other things that I've implemented as we dissect the solution, but these are probably the most important goals.

So, without further ado, here we go!

How We Will Pull It Off

In creating AJAX Warrior, I decided to go with "naked" Ajax, which means not using any library or toolkit at all. Developing games is usually a complex endeavor. I have written a number of games over the years, and I can say that they tend to be more complex than any of the enterprise-class development I do for a living. Because of this, I wanted to have the maximum degree of control over how things happened, and the best way to do that in my estimation is not to rely on any library.

Multipurpose Functions and Centralized Ajax

In exploring this application, you'll see a new technique to use when coding your own Ajax. This technique is a way to have a JavaScript function serve a dual purpose: to fire off an Ajax request, and to be its own callback. Let's jump directly into some code to see how this might work (Listing 10-1).

Listing 10-1. *An Example Ajax Function*

```
/**
 * Picks up an item the player is currently on.
 */
function pickUpItem() {

  if (xhr.request == null) {

    sendAJAX(pickUpItem, "pickUpItem.command", "", null);

  } else {

    if (xhr.json.iu == "true") {
      updatePlayerInfo(xhr.json.pn, xhr.json.ht, xhr.json.hp, xhr.json.gp);
    }
    // Always display the message.
    updateActivityScroll(xhr.json.mg);
    return true;

  } // End xhr.request == null if.

} // End pickUpItem().
```

Listing 10.1 shows one of the functions from the game code; specifically, the function is called when the player wants to pick up an item such as gold, spell scrolls, or health packs. We'll hold off on getting into the details of what is actually happening until later, but you need to recognize that here some branching is being done.

When the player wants to pick up an item, this function is called. It first checks to see if there is currently an Ajax request being processed by seeing whether or not `xhr.request` is `null` (again, do not get hung up on the details!). This would only be `null` if no Ajax request is currently in progress. In that case, it continues on and calls the `sendAJAX()` function, whose purpose I'm willing to bet you can guess! Note the first parameter passed to this function: it is a reference to the `pickUpItem()` function! This will be recorded as the desired callback function for the Ajax request.

Note that I said it will be recorded—it will not actually be registered with the `XMLHttpRequest` object associated with the request, as you might expect. Instead, `sendAJAX()` itself is the callback. To understand this, let's look at `sendAJAX()`, shown in Listing 10-2. However, I'm only going to show a trimmed version as the actual function is rather long and gets into details that we'll examine later. Listing 10.2 will give you the basic outline of its operation.

Listing 10-2. *The sendAJAX() Function (Trimmed Down)*

```
function sendAJAX(inCallback, inURL, inQueryString, inPostData) {

  if (xhr.request == null) {

    // Instantiate new XMLHttpRequest object.
    if (window.XMLHttpRequest) {
      xhr.request = new XMLHttpRequest();
    } else if (window.ActiveXObject) {
      xhr.request = new ActiveXObject("Microsoft.XMLHTTP");
    }

    // Make AJAX call.
    xhr.callback = inCallback;
    xhr.request.onreadystatechange = sendAJAX;

    // POST if inPostData is not null, GET otherwise.
    if (inPostData == null) {
      xhr.request.open("get", inURL + inQueryString, true);
    } else {
      xhr.request.open("post", inURL + inQueryString, true);
    }
    xhr.request.send(inPostData);

  } else {

    if (xhr.request.readyState == 4 && xhr.request.status == 200) {

      // Now call the callback function we recorded when initiating the
      // request.
      var clearVars = xhr.callback();

      // Finally, clear our variables associated with AJAX requests, if the
      // callback instructed us to (it wouldn't if it made another AJAX call).
      if (clearVars) {
        xhr.clearXHRVars();
      }

    } // End result status check.

  } // End XMLHttpRequest null check.

} // End sendAJAX().
```

So, let's follow the bouncing ball:

1. The player wants to pick up an item, so they press the Pick Up Item key, which calls `pickUpItem()`.
2. Assuming no other Ajax request is currently in progress, `sendAJAX()` is called.
3. `sendAJAX()` records the callback that `pickUpItems()` sent it, which is actually `pickUpItems()`, but it registers *itself* as the callback with the new `XMLHttpRequest` instance.
4. The request returns to `sendAJAX()`, and when a good response is received, that is, `readyState == 4` and `status == 200`, it calls the "real" callback, `pickUpItems()`.
5. `pickUpItems()` does its thing, and returns true.
6. Execution winds up back in `sendAJAX()`, where, seeing the result of calling the "real" callback was true, it nulls the variable holding the reference to the `XMLHttpRequest` object (`xhr.request`).

You may be asking yourself, "Isn't that a bit more complicated than it needs to be?" I do not believe so. There are two main benefits to this technique.

First, all of the actual Ajax code is centralized in `sendAJAX()`; it does not need to be duplicated anywhere else. Not only is this good in terms of code structure, it is also efficient because certain common things can be dealt with here instead of everywhere else—for instance, reacting to when the player dies. Instead of having to worry about all the various situations in which this could occur, we instead check for it in `sendAJAX()`. Since it can only occur as a result of some request to the server, it will be handled globally.

Second, this effectively eliminates concurrent Ajax requests, which could very well be a bad thing! Since all Ajax calls go through one function, and since this one function will only fire if another request is not already in progress, that problem is eliminated.

If both `pickUpItem()` and `sendAJAX()` checking to see if `xhr.request` is `null` seems redundant to you, just remember that both are Ajax callback functions. What would happen if we removed the check from `pickUpItems()`? We would not be able to differentiate between when we need to make the Ajax request—that is, when `xhr.request` is `null`—or when we need to handle the response—that is, when `xhr.request` is not `null`. What about if we remove it from `sendAJAX()`? In that case, we could again not determine if we can make an Ajax request, or whether we are being called as a result of a response returning from the server. It is not really about serializing Ajax requests, as you might initially expect, but that is a side effect, and fortunately, one we need anyway!

I hope you agree that this is a fairly elegant way to write Ajax code. I do not know if there is an actual name for this approach, but if not, feel free to refer to it as the Zammetti Approach!

JSON

At this point in this book, we've seen a number of ways to return data from the server from an Ajax request. We've seen XML. We've seen delimited strings. We've seen JavaScript being returned. We've even seen objects being returned (well, not really, but effectively that is what it looks like with DWR). For this project, we'll become familiar with another way to return data that is quickly becoming a big favorite of web developers: JSON.

JSON stands for JavaScript Object Notation. I feel this is a bit of a misnomer because while it *can* represent an object, it often does not. But that is just a name thing; the basic idea is that it is a way to structure data that is returned to a caller.

JSON is billed as being a lightweight, system-independent data interchange format that is easy for humans to read, easy for computers to parse, and easy for computers to generate. It uses a syntax that will be immediately familiar to most programmers who have any experience with a C-family language (including Java and JavaScript). It is built on two basic concepts that are pretty much universal in programming: a collection of name/value pairs, such as `Maps`, keyed lists, associative arrays, and so forth, and an order list of values, such as `Lists` or arrays.

Well, enough CompSci gobbly-gook! Let's see what JSON looks like:

```
{"firstName":"Frank","lastName":"Zammetti","age":"33"}
```

Really? Is that all there is to it? I wish I could try to impress you with my advanced knowledge, but no, that actually is all there is to it! As you can see, it looks similar to an array in Java, but not quite because two elements are defined between each delimiter. The item to the left of the colon is the key and the value to the right is the value. Each pair is separated by a comma, and the whole thing is wrapped in curly braces. Simple!

Where it gets really pretty cool is when you want to handle a JSON response in JavaScript. All you have to do is this:

```
eval("json = (" + xhr.responseText + ")");
```

The result, assuming `xhr` was the `XMLHttpRequest` that handled the response, is that a new variable, `json`, will be available to your script. From then on, if you want to get the first name in the response, you simply do this:

```
alert(json.firstName);
```

Really, that's it! The `eval()` call created the `json` variable, giving it the value of the response. The `json` variable is an associative array in JavaScript, so you can access the members just as you would any other associative array. Neat, huh?

You can send JSON to the server as well (although this project doesn't do that). If you go to `www.json.org/java/index.html`, you'll find some Java classes that help you generate and parse JSON. In this project, the only concern is generating JSON, and because again I wanted to have maximum control over the process, I wrote the code to do it myself. Of course, we're only talking about generating a string here; it certainly is not rocket science, as you'll see when we get to that code later.

I should mention that JSON is a general-purpose messaging format, and as such you can use it quite effectively outside Ajax work. Many people have taken to it much more than XML because it is less verbose but tends to be similarly human-readable. I'm sure we've all seen "bad" XML that is difficult to comprehend. Likewise, you can make JSON difficult to understand if you try.

Interestingly, to a certain extent, this project does just that that! For example, here's a real JSON response in AJAX Warrior:

```
{"dm":"false","pn":"Aragorn The Weak","ht":"100","hp":"1","gp":"10","iu":"true",
"vu":"true","di":"false","wn":"false","ec":"false","mo":"o","es":"false","md":"g
gggggggssss[[gggggggggggggggggg([[[[[[[[[[gggggggg[gggggggGgg[[[[ggggggggg[[[[
[(gggggg[[[[[[[[[[[[[[[[[[g^gggggg[[[[ggggggggg[[ggggggggggfgggggggggggfgggg
gggggggg"}
```

That does not look terribly readable to me! The names of the elements are obviously not meant for human consumption. Although you can probably guess quite a few of them, some you may not. The reason this is the case is that for a game, you want things to happen as quickly as possible in general. Therefore, I chose to make the JSON messages essentially unreadable to a human, who would likely never have to read them except perhaps for debugging purposes. So I made them as small and efficient as possible so as to (a) not take too long to generate or parse and (b) not take too long to transmit across the wire.

Most applications tend not to be quite as time-sensitive as a game, though, so I suggest always making your JSON (or XML for that matter!) as human-readable as possible. Saying `displayMessage` instead of `dm` and `playerName` instead of `pn`, for example, is what I recommend in such a case.

To learn more about JSON, check out the official JSON website, `www.json.org`. You'll find some reference materials and even code to help you work with JSON in a variety of languages, so if nothing else that might be worth it to you.

At this point, you are ready to use JSON, believe it or not! Go forth and be fruitful with your new knowledge!

With that out of the way, let's get into AJAX Warrior!

Visualizing the Finish Line

I suggest you spend some time playing AJAX Warrior before going forward. I hope you find it fun! It certainly is not something that would likely take more than an hour to finish, or even that long. There are a great many images I could show here, but because we have a lot of code to examine, I'm only going to show a handful of screens that are representative of the game. You'll see two pretty cool screens if you die or win the game, and they are not shown here. That should give you some incentive to play for a while!

The first screen I want to show you is the title screen, shown in Figure 10-1.

We have a nice little title banner up at the top, and an area for scrolling text. The player can switch this area between the story of the game, instructions, and some important notes (and they *are* important, so if you have not done so already, read them!). After that we have an entry box for the user to enter their name, as well as two buttons: one to start a new game, and one to continue an existing game.

Next up (Figure 10-2) is a shot that resembles what you'll see when AJAX Warrior begins. I say *resembles* because some of the setup of the game is random, so you may see something slightly different each time you start the game.

Figure 10-1. *The AJAX Warrior title screen*

Figure 10-2. *The player's first look at AJAX Warrior*

What does it look like when you are doing battle with a character in the game? Figure 10-3 provides the answer.

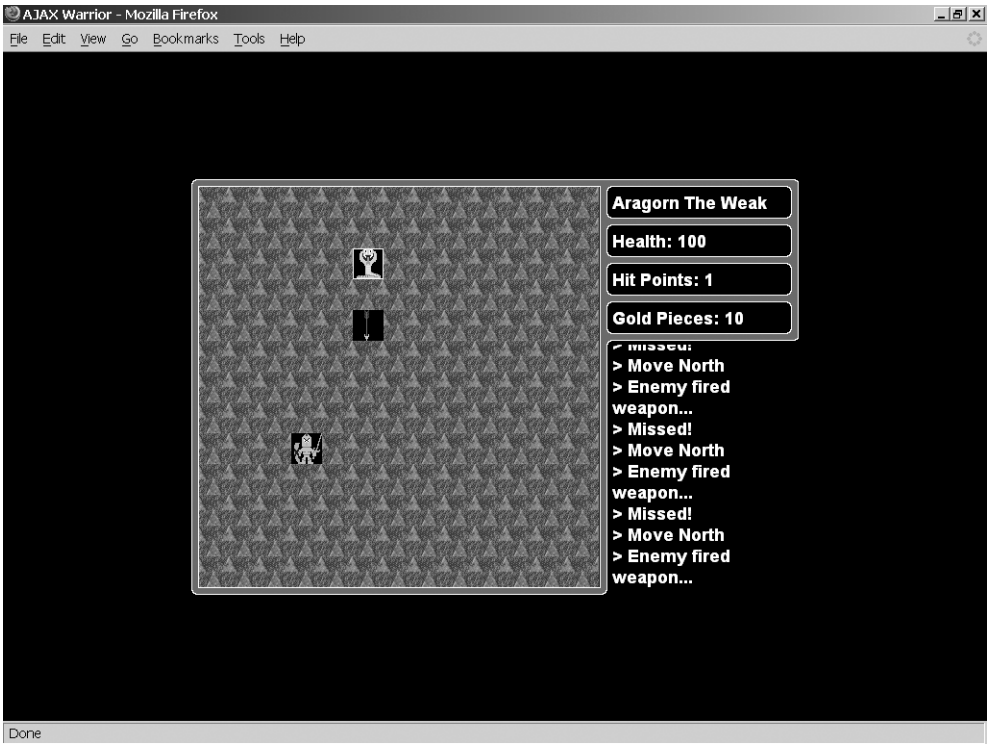

Figure 10-3. *Oh no, time to fight!*

One of the other activities you can engage in while playing is talking to characters, and in fact, this is a must because some will give you clues you'll need, and more important, some will provide you with items that are essential to winning the game. Figure 10-4 shows a conversation with a character.

Figure 10-4. *Hopefully he won't just want to talk about the weather!*

Lastly, although it is a somewhat pedestrian screen as compared to the others, Figure 10-5 shows what viewing your inventory looks like.

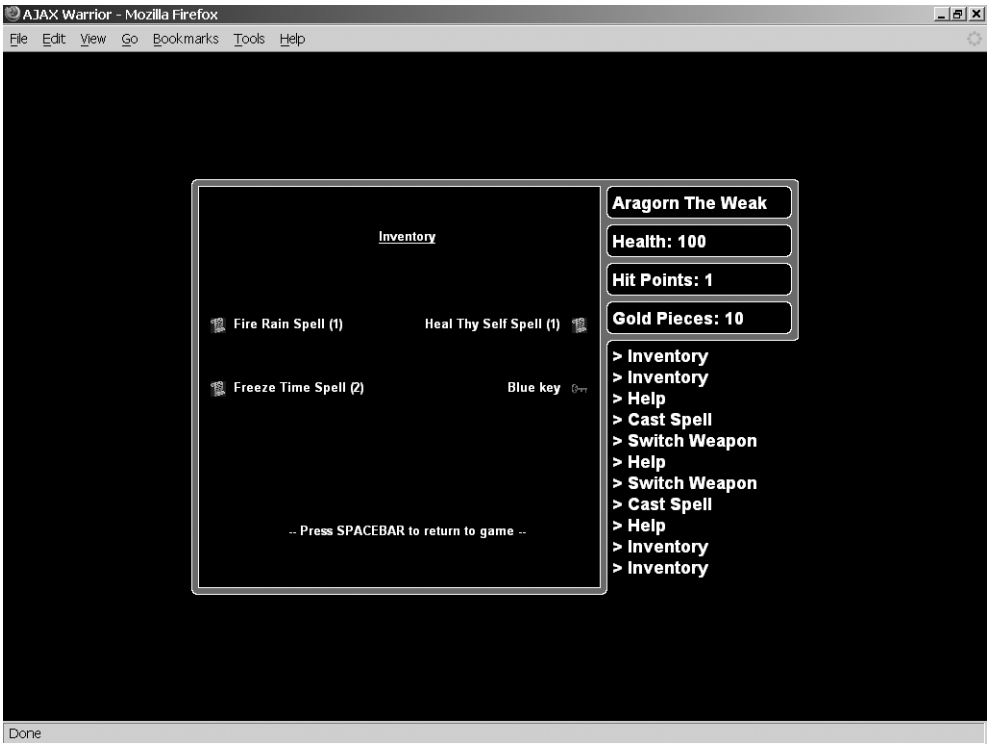

Figure 10-5. *Whaddaya got?*

Now that we know what AJAX Warrior looks like, we have only one small, minor, itsy-bitsy task to accomplish: tearing it apart and seeing what makes it tick!

Dissecting the Solution

First, please be sure to download the entire source for this project form the Apress website. Unlike many of the other projects in this book, I cannot list much of the source because this chapter is already rather long without it. It will therefore be important for you to have the source to look at as we go through the project.

Let's get a feel for the directory structure of AJAX Warrior. Unlike most of the other applications in the book, there is quite a lot to see here. Although it is still a typical webapp structure, there is more on top of that, and Figure 10-6 shows it. In this case, to conserve space, I have not expanded most of the branches to show the contents, so this is truly showing only the directory structure.

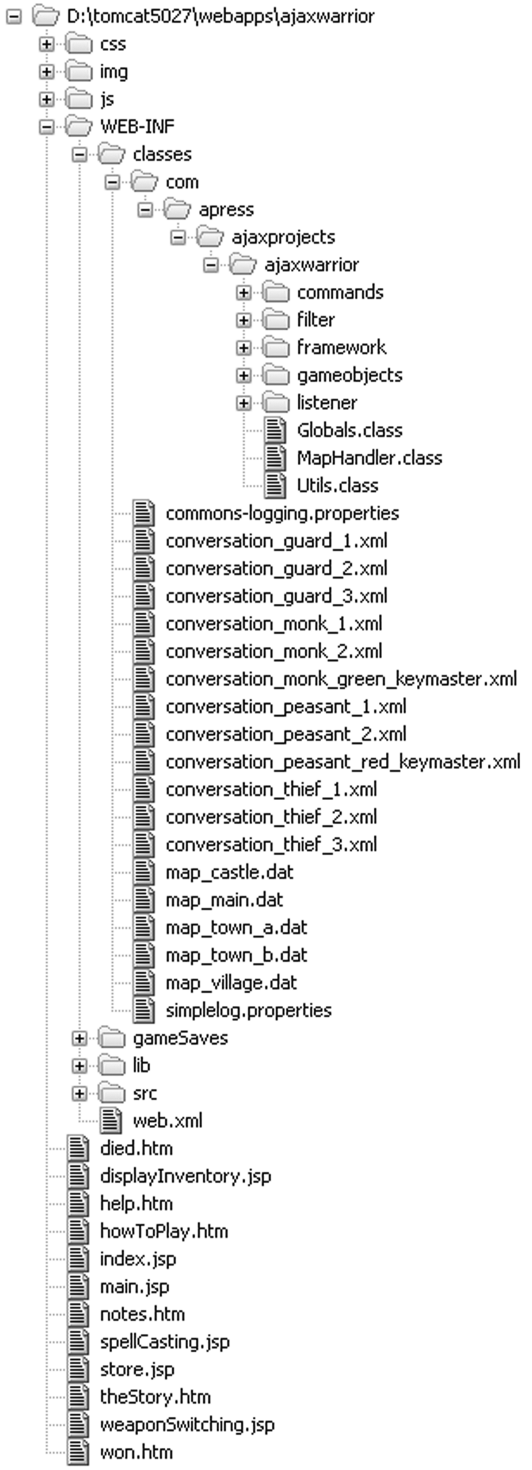

Figure 10-6. *Directory structure layout of AJAX Warrior*

At this point, the directory structure should be very familiar to you. In the root directory you'll find a number of HTML and JSP files, 13 of them. `index.jsp` is our welcome page. `main.jsp` is the actual game markup. All of the others are returned either as the result of an Ajax request (`died.htm`, `displayInventory.jsp`, `help.htm`, `spellCasting.jsp`, `store.jsp`, `weaponSwitching.jsp`, and `won.htm`). In the `/css` directory we find our typical single stylesheet, `styles.css`. In the `/img` directory are all the images for the application. The `/js` directory contains all the JavaScript for the game. We'll go over each one in some detail, and the same goes for the HTML and JSP files. However, Table 10-1 offers a breakdown of what the various JavaScript files contain.

Table 10-1. *Breakdown of the Numerous JavaScript Source Files in AJAX Warrior*

JavaScript File	Description
`ActivityScroll.js`	Contains code for working with the activity scroll (the area to the right below the player's information where messages are shown)
`BattleFuncs.js`	Contains code used when the player is fighting a character
`CastSpell.js`	Contains code used when the player is casting a spell
`Conversation.js`	Contains code used when the player is talking to a character
`GameFuncs.js`	Contains core game code—updating the map on the screen, for instance
`GameStateObject.js`	Contains a class that stores all the data defining the state of the game as far as the client side goes
`GlobalsObject.jsp`	Contains a class that houses constants and preloaded images
`Init.js`	Contains code that initializes the game
`KeyHandler.js`	Contains code that handles all keystrokes in the game
`SendAJAX.js`	Contains code that performs all Ajax requests throughout the game
`StoreFuncs.js`	Contains code used when the player is in a store
`SwitchWeapon.js`	Contains code used when the player wants to switch what weapon they are currently using
`UtilsObject.js`	Contains a handful of utility-type functions
`Vars.js`	Contains the few global (page-scoped) variables used in AJAX Warrior
`ViewChangeFuncs.js`	Contains code for switching between the various views in the game
`XHRObject.js`	Contains variables needed when making Ajax requests

After that we see the standard `WEB-INF` directory. There's nothing unusual, except you see a new directory: `/gameSaves`. As the name implies, this is where saved game data will be stored so that a player can continue a game later on.

Finally, the `WEB-INF/lib` folder contains all the libraries that AJAX Warrior depends on; they are listed in Table 10-2.

Table 10-2. *The JARs That AJAX Warrior Depends on, Found in WEB-INF\lib*

JAR	Description
`commons-logging-1.0.4.jar`	Jakarta Commons Logging is an abstraction layer that sits on top of a true logging implementation (like Log4J), which allows you to switch the underlying logging implementation without affecting your application code. It also provides a simple logger that outputs to `System.out`, which is what this application uses.
`commons-beanutils-1.7.0.jar`	The Jakarta Commons BeanUtils library, needed by Digester.
`commons-digester-1.7.jar`	Jakarta Commons Digester is a library for parsing XML and generating objects from it. This is used to parse some messages passed to the server by the client code.
`commons-lang-2.1.jar`	Jakarta Commons Lang are utility functions that enhance the Java language. Needed by Digester.
`javawebparts_request_v1.0_beta4.jar`	The Java Web Parts (JWP) request package; includes some useful utility classes for dealing with HTTP requests.
`javawebparts_core_v1.0_beta4.jar`	The JWP core package; required by all other JWP packages.

Before we start looking at the code, let me begin by saying that in this chapter I will rarely, if ever, list entire files, as I've tried to do throughout the rest of the book. One of my goals while writing this book was to make it so that you could be reading it without a computer in front of you and be able to understand what was going on. Therefore, I felt it was important to show complete listings as much as possible. I could not always do this; my editor had something to say about it! In this chapter, however, the decision was very easy: there's simply too much code, too many source files, for me to list them all. And once a few of them were not going to be listed, it was easy to decide to not list any of them in their entirety. Therefore, for this project, it is especially important that you download the source from the Apress website and follow along as you read.

Enough prefacing—let's get to it!

Of Maps and Conversations

The first topic I'd like to look at is not really code, but it is very important: maps and conversations. First, maps.

All of the maps used in AJAX Warrior are 100×100 elements. Each element is a character, and each character maps to a specific tile graphic (i.e., "m" is thin mountains, "w" is shallow water, and so on). The map files are stored in `WEB-INF/classes`, and so are accessible, as they are in the classpath. Printing a map here would be quite a waste of space, and would not look like anything but gibberish. However, I encourage you to look at one or two of them, and also look at the `Global.java` file to see what the various characters are.

Second, conversations. When the player talks to a character, it is not purely random, nor is it purely scripted. It is a web of conversation "nodes." Each node defines what the character says, and three replies the player can give. For each reply, the change in the character's karma

is stored, as well as the next node to jump to. The character's karma is important because as the player talks to the character, the character's karma goes up and down (or stays the same). If it reaches zero, the character is spooked and runs away. If it reaches 15, *and* the character is one of the two key masters, then they'll give the player the key (if they are not key masters, it doesn't really matter).

For each type of character that the player can talk to—guards, thieves, monks, and peasants—there are three unique conversation webs. One peasant and one monk is a key master. The conversations are stored as XML files, also in `WEB-INF/classes`. That XML looks like this:

```
<conversation id="thief_1">
  <node id="3" response="What do you want?">
    <reply id="1" karma="0" target="5">Nothing, I was just saying hello</reply>
    <reply id="2" karma="-2" target="10">I want to kill all thieves</reply>
    <reply id="3" karma="1" target="9">Umm, interesting conversation starter</reply>
  </node>
  <node id="4" response="That is none of my concern">
    <reply id="1" karma="-1" target="8">
      What I say should be of great concern
    </reply>
    <reply id="2" karma="0" target="14">I understand</reply>
    <reply id="3" karma="1" target="6">
      Well, let's talk about something else
    </reply>
  </node>
</conversation>
```

As you may have guessed, writing these XML files by hand is a bit tedious. So, in the source directory you'll find a Microsoft Excel spreadsheet. If you have Excel, load it up and play a bit. You'll find that each conversation is mapped out in a separate tab, and macros are mapped to buttons to validate the scripts and write out the XML. A script has to pass a number of validations, including making sure that there are no unreachable nodes; that no reply references its node; that there is a positive, negative, and neutral karma adjustment reply for each node; and that the character response and replies do not exceed a maximum length.

Make no mistake; it still can be a bit tedious to write a script, even with this spreadsheet. But it is considerably easier than doing it manually, and the validations ensure that the scripts will make some kind of sense!

The Client-Side Code

Although it is not strictly speaking client-side code, let's begin by looking at `web.xml`, just to be sure there is nothing fishy going on there—and as it turns out, there's not! We have our typical welcome page defined as `index.jsp`, and a session timeout value set to 60 minutes. We also see a single servlet defined:

```
  <servlet>
    <servlet-name>FrontServlet</servlet-name>
    <servlet-class>
      com.apress.ajaxprojects.ajaxwarrior.framework.FrontServlet
```

```
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>FrontServlet</servlet-name>
    <url-pattern>*.command</url-pattern>
  </servlet-mapping>
```

All of the requests made during our game—all our Ajax requests—will go through this servlet, allowing us to have a centralized place to handle some common functions, as we'll see later.

We then see a single filter defined:

```
  <filter>
    <filter-name>sessionCheckerFilter</filter-name>
    <filter-class>
      com.apress.ajaxprojects.ajaxwarrior.filter.SessionCheckerFilter
    </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>sessionCheckerFilter</filter-name>
    <url-pattern>*.command</url-pattern>
  </filter-mapping>
```

Although we'll see it later, the purpose of this filter is to be sure that a game has been correctly started for any request that passes through our servlet. If a game has not been properly started, the request is directed back to `index.jsp`. All our requests will end with `.command`, which is what we map the servlet to.

Lastly, we have a single context listener:

```
  <listener>
      <listener-class>
        com.apress.ajaxprojects.ajaxwarrior.listener.ContextListener
      </listener-class>
  </listener>
```

This listener will handle any server-side application initialization that should occur at startup. Again, we'll see this in detail later.

index.jsp

The next thing to examine is `index.jsp`. This is the initial page the player sees, the one with the AJAX Warrior title banner and the scrolling area. The first thing found in the code is some image preloads for the buttons you see on the screen. There are two images for each button: a normal version and the version seen when you hover over the button with the mouse. After that is a batch of variables beginning with `vs_`, which are used for the vertical scroller in the middle of the page.

The first executable code we see is the `init()` function, which is called in response to the page's `onLoad` event:

```
function init() {

  vs_contain = document.getElementById("vs_container");
  vs_content = document.getElementById("vs_contents");
  layerCenterH(vs_contain);
  layerCenterH(document.getElementById("controls"));
  <% if (request.getAttribute("Error") != null) { %>
  alert("<%=request.getAttribute("Error")%>");
  <% } %>
  switchContents("theStory");

} // End init().
```

First, references are grabbed to two elements: vs_container and vs_contents. Note that is used instead of to avoid the line break that puts after itself. The vs_ portion stands for (v)ertical (s)croll, and vs_container is the with the gray background and is the area the scroller takes up. The vs_contents is where the actual contents that will be scrolled go. Note that these two reference variables are page-scoped and are used throughout the rest of the page.

After that, we use the layerCenterH() function that we have previously seen in a number of chapters to center the vertical scroller. The same is done for the controls , which contains all our buttons and a text box for entering a name.

After that we have a JSP scriptlet that renders an alert if an attribute named Error is found in the request. This will be present if the name the player enters is already in use, or if they tried to continue a game that does not exist.

Finally, switchContents() is called, passing in the ID of a that contains text we want to scroll:

```
function switchContents(inWhichText) {
  stopScroller();
  vs_content.innerHTML = document.getElementById(inWhichText).innerHTML;
  resetScroller();
  startScroller();
} // End switchScrollText();
```

As you can see, the scroll is first stopped by calling stopScroller(), and the contents of the that vs_content points to are updated with the contents of the passed in. Then the scroller is reset by calling resetScroller() (so that if it was previously scrolling it will start from the beginning of the new content), and it is then started again by calling startScroller(). switchContents() is called when the user clicks The Story, How To Play, or Important Notes button.

Speaking of resetScroller(), let's see what it looks like:

```
function resetScroller() {
  // Determine milliseconds
  vs_milliseconds = 1000 / vs_scroll_speed;
  // Get height of container
  vs_container_height = vs_contain.style.height.substr(0,
    vs_contain.style.height.length - 2);
```

```
  // Get height of contents
  vs_contents_height = vs_content.scrollHeight;
  // Start off bottom
  vs_contents_top = (1 * vs_container_height) + 20;
  vs_content.style.top = vs_contents_top;
  // Make contents visibile
  vs_content.style.visibility = "visible";
} // End resetScroller().
```

The variable `vs_milliseconds` specifies how many times per second the timer that causes the contents to scroll will be fired. `vs_scroll_speed` specifies how many lines the contents should scroll up per second, so dividing 1,000 by this number gives us the number of milliseconds required between each move (since there are 1,000 milliseconds in a second). `vs_container_height` is the height of the container `<span>`, and likewise, `vs_contents_height` is the height of the actual contents to be scrolled, both of which are needed to determine when scrolling has completed and the contents should recycle and scroll again. Note that the value returned by getting the value `style.height` is in the form `99px`, where 99 is the actual height. So, we need to strip the `px` portion since we just want the number, hence the use of the `substr()` function. `vs_contents_top` is a variable that controls the value of the `top` style attribute. The way the scroll works is quite simple: the `vs_contents` `<span>` is set up to clip the contents, which means that scroll bars will not be present if the contents of the `<span>` are larger than the span itself, and the `<span>` will not resize to accommodate the contents. So, if we set the value of the `top` style attribute of the contents to something larger than the height of the container `<span>`, the contents won't be visible. If we then slowly subtract from that `top` value, the contents will slowly scroll up from the bottom of the container `<span>`. If we keep doing this until the contents have completely scrolled up (which means the `top` style attribute will be a negative value at that point, which is perfectly valid), we have ourselves a vertical scroller!

The `startScroller()` function is literally only this line:

```
vs_interval_id = setInterval('doScroller()', vs_milliseconds);
```

This simply sets up a timeout that fires after the amount of milliseconds determined in `resetScroller()`. As you can probably guess, `stopScroller()` is nothing but clearing this timeout, and also hiding the contents of the scroller.

The last scroller-related function is the target of the timeout, `doScroller()`, and I think you'll be surprised at how simplistic it is:

```
function doScroller() {
  // Only do this if we're not paused
  if (!vs_pause) {
    // Move up one pixel
    vs_contents_top--;
    // If we've scrolled off the top, reset to off the bottom
    if (vs_contents_top < -vs_contents_height) {
      vs_contents_top = (1 * vs_container_height) + 20;
    }
    // Reposition contents layer
    vs_content.style.top = vs_contents_top;
  }
} // End doScroller().
```

Yes, that is indeed it! We first check the value of the `vs_pause` variable, which is set to true when the player hovers over the scroller. This allows the player to pause the scroller to read it. If that variable is false, though, we simply subtract 1 from the `vs_contents_top` variable. We then check to see if we've scrolled all the way, which is done by comparing the value of `vs_contents_top` with the height of the contents stored in the variable `vs_contents_height`. When the former is less than negative the height, the scroll is complete. To make that a bit clearer, let's say the height of the contents is 200 pixels. When the `top` style attribute of the contents is less than –200, the scroll has completed. Remember, the contents of the container `<span>` will clip, so that if the contents are –200 pixels above the top of the container, it is no longer visible. At that point, the `top` attribute is reset to essentially push the contents down below the container, and finally, the `top` style attribute of the contents `<span>` is updated with the new value of the `vs_contents_top` variable.

The markup on the page is quite simple as well. We have a `<div>` with the ID `pleaseWait`. This is displayed when the user clicks New Game or Continue Game. This is done to avoid some JavaScript errors that can occur because of the image rollovers on the buttons. In essence, the page will be overwritten in memory with the actual game, but the screen will not immediately be updated. This means that if you roll over a button, the JavaScript that handled the rollover will have been overwritten, and an error occurs. The error is actually "invisible" to the user, but I did not like seeing it showing up even just in the debugger in Firefox, and this gets around it.

Following that is the AJAX Warrior title banner. Immediately after that are the two `<span>` elements for the vertical scroller. Note the `onMouseOver()` handler on the container; this handler is used to pause the scroller.

After that is our control `<div>`. This contains the buttons, as well as the text box for entering the player's name. The latter is part of a form that is submitted to `startGame.command`. Note the hidden `whatFunction` field. This will be populated either with the value "newGame" or "continueGame", depending on which button the player clicks. When the form is submitted, the `checkName()` function is called. This verifies that something was entered, and that it does not contain any invalid characters (since this will be a filename, only numbers, letters, dashes, underscores, and spaces are allowed).

Following that are three JSP includes: one for the contents of The Story, one for the contents of How To Play, and one for the contents of Important Notes. The files included are, unimaginatively, `theStory.htm`, `howToPlay.htm`, and `notes.htm`. Each is nothing but plain text wrapped in a `<div>`. Have a peek if you don't believe me!

main.jsp

Once the form is submitted and the server determines the correct outcome, either `index.jsp` will be shown again (if an error occurred, such as the game the player wants to continue cannot be found) or `main.jsp` will be shown. Let's now jump right into `main.jsp`. Refer to the listing for this JSP that you've downloaded from the Apress website.

There really is not any actual code here, just a whole batch of JavaScript imports... a JSP scriptlet wrapped in a JavaScript `<script>` block (I'll explain this in just a moment)... and finally, a whole bunch of `<div>` elements.

Let's jump back to that scriptlet for a moment. We'll learn shortly that there is a `GameState` object that stores, well, information about the current state of the game. In fact, there are *two* such objects, one server side and one client side. They both store different sets of information, but to properly save and restore a game, we need to save and restore both objects.

When `main.jsp` loads, recall that it could be as a result of a new game being started or an existing game being continued. In the latter case, we'll find that a serialized version of the client-side `GameState` object will have been put in request as an attribute under the name `clientSideGameState`. To continue the game, we need to reconstitute this serialized version into a real `GameState` object. The details of that reconstitution will be explored when we get to the JavaScript source, but in short, the string is put into a page-scoped variable named `clientSideGameState`. This variable is then passed to the `init()` function as a result of the `onLoad` event handler.

As for the `<div>` elements, `divGame` is the entire game area. Everything else is contained with this element. `divBorder` is the container of the border image. `divMap` is the actual gameplay area. `imgCharacter`, which is an `<img>` tag and not a `<div>`, is where the close-up of a character the player is talking to appears. `divTalkingReplies` is the blue box superimposed over the character listing the replies the player can give during conversation. `divInventory`, `divSpellCasting`, `divWeaponSwitching`, `divStore`, `divHelp`, and `divGameEnd` are areas that will have dynamic content placed (except `divHelp`, `divGameEnd`, and `divStore`, which are static) and will obscure the gameplay area. `divName`, `divHealth`, `divHitPoints`, and `divGoldPieces` are areas that will display the player's current information on the right. Finally, `divActivityScroll` is the area to the right below the player's information where messages are displayed.

styles.css

Note that all of these have a specific style class applied, and those classes are found in `styles.css`. They define font styles and colors and such, but also define positioning for the elements. All of the attributes used should by now be quite familiar to you; nothing fancy is going on. The one aspect that deserves some discussion is positioning.

It's important to remember that all of these are children of the `divGame` element. So, when we position another element within that one using the `absolute` value for the `position` attribute, it is an absolute value that is relative to the containing element. That may be a bit confusing, but you have to think like Einstein in terms of relativity. Usually, when absolute positioning is used, it is absolute relative to the page itself, which makes it seem really absolute, but in fact it is still relative.

The positioning may be easier to comprehend if you see it graphically. Figure 10-7 shows an exploded view of where the various layers get positioned. I use the border as the reference because in reality, all of the positioning is based on fitting into specific areas of the border. Recall, however, that the border is not the outer element; it is a container within a `<div>`. But the border is absolutely positioned within that container and is sized to fill the entire container; therefore, for all intents and purposes, you can think of all the other `<div>` elements as positioned relative to the border, as the diagram shows.

If you are uncertain about all this, an easy way to see is to add `display:none;` to the `cssBorder` class in `styles.css` and then start the game. With the border not visible, note that all the other elements are still where they should be. Again, because they are positioned absolutely within the `divGame` container, not with regard to the border, they appear to be positioned relative to the border.

Figure 10-7. *How the various elements relate to one another positionally*

died.jsp

Well, now, that is quite a dreary heading! This HTML document is displayed when the player dies (or when they kill a key master, since the game cannot be won at that point). Listing 10-3 contains the entire contents of this file.

Listing 10-3. *died.htm, in its Entirety*

```
<img src="img/game_lost.gif">
<div style="position:absolute;top:130px;left:2px;width:416px;height:100px;">
  <center>
    Thou art dead!
    <br>
    Please do try again!
    <br><br>
    Press any key to go to start screen.
  </center>
</div>
```

What happens is that with each of our JSON responses from the server, two elements are present: `di` and `wn`. `di` will be true if the player died; `wn` will be true if the player won the game (regardless of what the request was, since at least dying can happen in multiple ways). When the JavaScript function that sends Ajax requests (which we'll explore in a moment) sees that `di` is true in the response, it immediately sends another Ajax request targeting `died.htm`. The contents of this file are used to populate the `divGameEnd <div>` in `main.jsp`, and those contents are then displayed. This process is identical for when the player wins the game, and the markup is essentially identical to that shown in Listing 10-3, with a different image and different text, of course.

displayInventory.jsp

This file is a bit meatier. Here we're constructing markup to display the player's inventory. The markup is constructed by first getting the `GameState` object instance from request, and then getting the inventory collection from it. For now, it is enough to understand that the `GameState` object is an object on the server that stores all the information about the current state of the game. Things like the player's name and health, their inventory, what weapon they are currently using, and so on are found there. Recall that earlier I mentioned that there is a `GameState` object on both the client and server. The server-side object contains much more information, but both objects serve the same basic purpose.

Once we have the inventory collection, it is first checked to see if it is empty. If it is, some simple markup telling the user they are holding nothing is rendered. If it is not empty, though, the code begins to iterate over the collection. For each item, we determine what it is and output the appropriate markup. For instance, if it is the blue key, we see this:

```
case Globals.ITEM_KEY_BLUE:
  out.print("<img src=\"img/item_key_blue.gif\" " +
    "align=\"absmiddle\" width=\"16\" height=\"16\">");
break;
```

`Globals` is a class that contains a large number of constants used throughout the code. One of them is the code representing the blue key.

Note that the rendered markup is forming a table with two columns. So, we need to keep track of whether the item we're adding is in the first or second column so that we can end the row properly when the time comes. The `firstColumn` variable is used to keep track of that.

For each item in the inventory, we make a call to `Utils.getDescFromCode()`. `Utils` is a class that contains a handful of utility functions, `getDescFromCode()` among them. This function returns a descriptive string for the item code passed in; for instance, if we pass it `A`, which is the code for the blue key, it will return `Blue key`.

help.htm

This is the help screen the player sees when they press the H key during play. It is a pretty pedestrian piece of code and is just plain old HTML. I therefore leave this in your capable hands to check out.

spellCasting.jsp

This is the page that we display when the player wants to cast a spell. It is conceptually (and even structurally) very similar to `displayInventory.jsp`. Like that JSP, we get the inventory of the player from `GameState`, and check to see if it's empty. If it is, we render the markup to tell that to the player. If it isn't, we begin to iterate over it.

This time around, we know that there are only three possible spells, and we want to have specific keys the user can press for each to cast them. So, we switch on the inventory code and display the appropriate text for the spells only. We must also keep track of which spells the player has. Think of it this way: there is some JavaScript floating around somewhere that we'll see shortly that handles key presses. When the spell casting display is showing, the player can press F, H, or T, corresponding to the spells they can cast. However, how does that JavaScript know which of those is valid, because remember, the player may not have any spells, or may

not have them all? It would be nice to not have to go to the server just to find out the player doesn't have a spell they requested casting. So, we construct a list of the spells the player has as we render this markup. This is a space-separated list of the codes for the spells. So, in other words, if the player had all the spells, we would get the string "` ; . dummy"; each of those is the code for a given spell (the codes will become a bit clearer later—just go with the flow for now!). At the end of string we also put "dummy", so that when the string is tokenized, there is always a value at the end and not a delimiter, which would cause a problem.

This is where it gets a bit interesting: at the very end of `spellCasting.jsp`, we see this line:

```
<script>gameState.spellsPlayerHas = "<%=spellsPlayerHas%>";</script>
```

Recall that this page will be rendered and returned as the result of an Ajax call. The JavaScript making that call looks for a script block in the response, and if found, `eval()`'s it. So, in this case, the result is that the `spellsPlayerHas` field of the client-side `gameState` object will be populated with the string that was constructed listing what spells the player has. So, that field can then be used when the player presses a key, say T, to see if they have that spell. If not, the keystroke is ignored.

store.jsp

The player can enter a store in the two towns and purchase various items. This JSP is the markup that is displayed when the player is in the store. It is nothing but a `<table>` listing all the items and how much they cost. The costs are taken from the `Globals` object. Aside from that, there's not much to see here, so have a quick peek and let's move along.

weaponSwitching.jsp

This is the file that renders the markup seen when the player wants to switch weapons. It is once again very much along the same lines as `spellCasting.jsp` and `displayInventory.jsp`. The only real difference is that because "Bare hands" is always an available option, there is never the possibility of inventory being empty or the player not having any spells to cast, as in the other two JSPs. So, there is no branch checking for emptiness here. Aside from that, it is very similar. Again, we have a string of weapons the player has built up, and again, a `<script>` block at the end is rendered and will be `eval()`'d to get the value into the `gameState` object to use when keystrokes are handled.

And with that, we have seen all of the markup for AJAX Warrior, and we've examined the stylesheet used. Now let's move on to the JavaScript, where most of the action is.

Globals.js

The first JavaScript file I'd like to discuss is `GlobalsObject.js`. This file contains the definition of a single JavaScript class, `GlobalsObject`. This is similar to the `Globals` class on the server in that it stores some constants used throughout the code. The vast majority of what is in this object are preloaded images. Note that the extension of this file is `.jsp` and not `.js` as the rest are. The reason is that we need to reference the values in the `Globals` class on the server, and we could not do that unless this was a JSP. The container will kindly evaluate the JSP for us, even when a `<script>` tag on a page includes it.

Let's look at a snippet of the beginning of this code:

```
function GlobalsObject() {

  // Viewport Sizes.
  this.TILE_WIDTH = <%=Globals.TILE_WIDTH%>;
  this.TILE_HEIGHT = <%=Globals.TILE_HEIGHT%>;
  this.VIEWPORT_WIDTH = <%=Globals.VIEWPORT_WIDTH%>;
  this.VIEWPORT_HEIGHT = <%=Globals.VIEWPORT_HEIGHT%>;
  this.VIEWPORT_HALF_WIDTH = <%=Globals.VIEWPORT_HALF_WIDTH%>;
  this.VIEWPORT_HALF_HEIGHT = <%=Globals.VIEWPORT_HALF_HEIGHT%>;
```

As you are well aware by now, we create a class in JavaScript by creating a function. Inside it we can define fields by using the `this` keyword. So here we are adding some fields: `TILE_WIDTH`, `TILE_HEIGHT`, `VIEWPORT_WIDTH`, `VIEWPORT_HEIGHT`, `VIEWPORT_HALF_WIDTH`, and `VIEWPORT_HALF_HEIGHT`. The values for these fields are all taken from the server-side `Globals` class (they are static finals). Since there is no such thing as final in JavaScript, we cannot get the true constants effect here; these fields are still alterable. We just have to hope the programmer is smart enough to not do so.

There are constants here mimicking almost all of the values in the server-side `Globals` class. As I mentioned earlier, most of the contents are image preloads. For instance:

```
this.imgITEM_GOLD = new Image(<%=Globals.TILE_WIDTH%>, <%=Globals.TILE_HEIGHT%>);
this.imgITEM_GOLD.src = "img/item_gold.gif";
```

This is the preloaded image for the chest of gold the player can pick up. We also see some constants defined for key handling, like so:

```
this.KEY_SPACEBAR = 32;
this.KEY_LEFT_ARROW = 37;
this.KEY_RIGHT_ARROW = 39;
this.KEY_UP_ARROW = 38;
this.KEY_DOWN_ARROW = 40;
```

The numbers are the key codes that will be received in our keystroke event handlers when the player presses a key. Better to reference these constants throughout the code than the numeric values themselves!

Init.js

The `init()` function, contained in `init.js`, is called `onLoad` of `main.jsp`. It is responsible for initialization tasks for the game. These include

- Instantiating a number of objects such as `GameStateObject` and `GlobalsObject`
- Clearing out the activity scroll (the area to the right where messages appear)
- Reconstituting the `GameStateObject` instance if a game is being continued (in fact, the object referenced by the `gameState` variable itself does this, but `init()` makes the decision whether to ask the object to do it)
- Centering the game in the window
- Creating the 169 images that are the tiles our map display is built from

- Hooking the keyUp event handler so we can process key presses
- Hiding the various secondary displays (inventory, spell casting, help, weapon switching, and the store) since they are not hidden initially
- Making the initial Ajax request to display the map

I think that for the most part this code is pretty self-explanatory, but one part is worth a second look—the code that creates those map tile images:

```
var x;
var y;
for (y = 0; y < Globals.VIEWPORT_HEIGHT; y++) {
  for (x = 0; x < Globals.VIEWPORT_WIDTH; x++) {
    var newImg = document.createElement("img");
    newImg.style.position = "absolute";
    newImg.style.left = (x * Globals.TILE_WIDTH) + "px";
    newImg.style.top = (y * Globals.TILE_HEIGHT) + "px";
    newImg.width = Globals.TILE_WIDTH;
    newImg.height = Globals.TILE_HEIGHT;
    newImg.id = "tile-" + y + "-" + x;
    var map = document.getElementById("divMap");
    map.appendChild(newImg);
    document.getElementById("tile-" + y + "-" + x).src =
      Globals.imgTILE_BLANK.src;
  }
}
```

When you're playing the game and you see the map, you're seeing a viewport on a larger world. The map for the entire world of Xandor, for instance, is 100 tiles wide by 100 tiles high. However, you only see 13×13 of that at a time. The viewport is a grid of 13×13 images (169 in total). So, to display a viewport on the map, we get a chunk of the map that is 13 lines tall and 13 characters wide, where each character is a code representing a specific tile type (i.e., mountains, water, a town). So, the bottom line here is we need 169 images on the page whose src attribute we can update to reflect the tile that should currently be shown in it to form the viewport on the map.

However, it would be unwieldy to actually have 169 <img> tags on the page. Instead, we use some DOM functions to create them. We begin with a loop, which iterates the number of times there are lines in the viewport (13). Another loop inside that iterates for each character in the row (13). For each, we create a new <img> object. We set its position style attribute to absolute, and set its width and height to the width and height defined in Globals for a tile (32×32 pixels). We then set its left and top positions, using the value of the loops to form a grid. We then give it an ID formed by taking the string "tile-" and appending the y loop value and x loop value, separated by a dash. Finally, we get a reference to the divMap , and append the new image object as a child of it. Finally, we get a reference to the <img> we just added and set its src to our blank tile. And that's how we get a grid of images to create our map viewport.

GameState.js

I have mentioned this `GameState` object thing a few times, and now it is time to check it out.

The `GameStateObject` is, by and large, just a JavaBean, or what would be the equivalent of a JavaBean in JavaScript. We have some fields here: `activityScroll`, which is an array that contains the messages seen in the activity scroll to the right; `currentView`, which is a reference to the `<div>` we are currently seeing (`divMap`, `divInventory`, etc.); and `previousView`, which is used when we view inventory, help, spell casting, or weapons switching, so we know whether we should show the map again, or whether we were in the middle of battle and should show the battle view again. `spellCast` and `weaponSwitched` are two simple true/false flags used to determine when a spell was just cast or when a weapon was just switched. `weaponsPlayerHas` and `spellsPlayerHas` are the strings generated by `weaponSwitching.jsp` or `spellCasting.jsp` to tell us which weapons or spells the player currently has. `talkAttackMode` determines whether the player is currently in Talk mode (blue border) or Attack mode (red border, which means the player will attack any character they come in contact with). `currentWeapon` specifies which weapon the player is currently using. `fireProjectile`, similar to `spellCast` and `weaponSwitched`, tells when a projectile weapon (slingshot or crossbow) has been fired. All of the variables prefixed with "projectile" are used when an arrow is flying either from the player or from a character. `battleEnemyTurn` is another flag that is set to true when in battle it is the character's turn to move.

Next we have the `serialize()` function. This is used when the player requests that the game be saved. This function constructs some XML representing the current `gameState` object. However, as it turns out, it's not important to save *all* the data contained in this object, so we ignore the unimportant fields and only serialize what we absolutely have to in order to persist the state of the game. This is equivalent to marking a field transient in Java, but since there is no notion of transient fields in JavaScript (mostly because there is no inherent notion of serialization), we simply ignore what we do not need to save.

To go along with `serialize()` is `reconstitute()`, which is called when a game is loaded (as a result of a call from that branching logic in `Init.js` we saw earlier). Recall that what the server returns to us is the same data that `serialize()` constructed, but as a delimited string (delimited by ~~~). So, we split this string, and set the fields of the `gameState` object based on its values. No big deal.

Utils.js

`Utils.js` contains a single class, `UtilsObject`, which itself contains two functions that we have previously seen: `layerCenterH()` and `layerCenterV()`. Although this is the first time we've seen them as two functions like this, we have in fact seen and examined the code they contain: in InstaMail in Chapter 5 and PhotoShare in Chapter 7. In those chapters, we combined the code and used it to center the Please Wait layers. Here, I've broken them out into two separate callable functions for more flexibility. Both functions accept a reference to some element on a page, and they then center that element, horizontal or vertically as applicable. Since we reviewed those functions in previous chapters, I will not go into detail here.

Vars.js

One of the things I wanted to demonstrate with this application is the concept of not polluting global namespace, or not using a lot of global variables. When you're trying to write more

robust, professional-quality JavaScript, it is a good idea, as it is in Java (although you do not have a choice in Java as you do in JavaScript) to deal in objects as much as possible. So, instead of having the functions in `Utils.js` in page scope, for instance, I created a `UtilsObject` to house them. The benefit to this is that it avoids naming conflicts. If you wanted to have `layerCenterH()` at page scope, you could; the two would not conflict.

However, it is virtually impossible for an application of this complexity to not have *some* global variables, and indeed we have a few. But very few indeed:

```
var Globals = null;
var Utils = null;
var gameState = null;
var xhr = null;
```

That is the entire contents of `Vars.js`, minus the comments. `Globals` is a reference to the instance of the `GlobalsObject` class. `Utils` is a reference to the `UtilsObject` class. `gameState` is a reference to the `GameStateObject` class, and `xhr` is a reference to the `XHRObject` class.

XHRObject.js

Speaking of the `XHRObject`, we now come to `XHRObject.js`, which defines that class. Listing 10-4 shows the entire contents of this file. (Sorry, I said I wouldn't do this too often!)

Listing 10-4. *Against My Own Rule, the Entire XHRObject.js File*

```
/**
* This object contains three variables used to make Ajax requests.  The member
 * variable request is actually the XMLHttpRequest instance.  The callback
 * member is the function that is the callback for the current Ajax request.
 * The json member is the parsed JSON response.
 */
function XHRObject() {

  this.request = null;
  this.callback = null;
  this.json = null;

  /**
   * Function to null our XMLHttpRequest-related vars.
   */
  this.clearXHRVars = function() {
    this.callback = null;
    this.json = null;
    this.request = null;
  } // End clearXHRVars().

} // End XHRObject.
```

As the comments say, this object contains some fields used during Ajax request sending. `request` is literally the current `XMLHttpRequest` instance, `callback` is a reference to the function that is the callback for the request, and `json` is the parsed JSON response from the server.

SendAJAX.js

Having seen the `XHRObject` class, let's now see what makes use of it: the `sendAJAX()` function contained in `SendAJAX.js`. This function is used by all the other game code to make Ajax requests.

`sendAJAX()` accepts a number of parameters. First, it accepts a reference to the function that will be the callback. This will nearly always be the function that called `sendAJAX()`. It also accepts, as you would expect, the URL to submit the request to. It accepts both a query string (fully formed) and POST data. It technically will accept and use both; however, the call method (GET or POST) is determined by the presence of post, so if you send POST data to `sendAJAX()`, the method will be POST, regardless of whether or not there is a query string (but the query string *will* be used even if POSTing).

As we discussed in the "How We Will Pull It Off" section earlier, this function sets itself up as the callback in the newly instantiated `XMLHttpRequest`. So, first it checks whether there is already an `XMLHttpRequest` object reference in `gameState` (the `xhr` field). If so, then `sendAJAX()` proceeds (via an `if` branch) to check the status of the request. When that request completes, the results sent back by the server will be processed.

If there is no existing request in progress, the function instantiates a new `XMLHttpRequest` object, and attaches a parameter to the query string that has a value of the current date/time in milliseconds. This ensures that the browser will not cache the response and thereby cause it to appear as if the server is not responding (which it isn't because the request would never have reached the server). It then sets the method based on whether or not `inPostData` is `null`, and fires off the request.

When the response is received, the `else` part of the branch logic is fired, continually of course, until the response is good. When it is, the response is first `eval()`'d into `xhr.json`, which we'll reference from then on.

A number of various checks are then performed. If the `ex` member is present in `xhr.json`, then an exception occurred on the server and an alert box is displayed to let the player know. If the `di` member is set to true in `xhr.json`, then the player died, and the player's information is updated (so that their health can be shown as 0). Also, the activity scroll is updated so that the final message from the server is seen, and the game end screen is shown via the line

```
showGameEnd("died");
```

`showGameEnd()` accepts the values "died" and "won", and loads the appropriate content (`died.htm` or `won.htm`).

If the `ct` member is present in `xhr.json`, that means we are beginning to talk to a character, so the `startTalking()` function is called to do that (we'll see that in a bit).

If the `mo` member in `xhr.json` indicates we are now in Battle mode, and the current view is *not* Battle mode, that means we need to switch to Battle mode now. We do this by setting the current view to `battle`, and calling `updateMap()`, which will redraw the screen in Battle mode.

Lastly, if the `mo` member in `xhr.json` indicates we are not in normal (i.e., walking around the map) mode, and the current mode on the client is Battle, then we are coming out of battle, which again requires setting the current view and calling `updateMap()`.

Note that each of these checks results in returning from the function. Before that, however, a call to `xhr.clearXHRVars();` is made, which `null`s out the `xhr.request`, `xhr.json`, and `xhr.callback` fields. It is only at this point that another Ajax request could occur (recall that the check to see if `xhr.request` is `null` is the first thing this function does).

Now, if none of these conditions applied, then `sendAJAX()` needs to call on the original callback that was passed into it. It does so, and captures the result. If the result is true, then `xhr.clearXHRVars()` is called. If false, it does not make that call. That way, if the callback function itself wants to make an Ajax request, it can do so. Think about what would happen if `xhr.clearXHRVars()` was called in that case—it would step on that new Ajax request and the request would not go through. Fortunately, this situation only comes up a few times, a majority of the time the callback will return true.

ActivityScroll.js

This source file contains a single function, `updateActivityScroll()`, which is called any time a new message should be displayed to the user in the activity scroll on the right side.

The activity scroll itself, as you might recall from our discussion of `GameStateObject`, is just an array. It always has 11 elements, which is how many can fit in the area allotted on the screen for messages. The messages are displayed top to bottom. In other words, each time the activity scroll is updated, it is redisplayed in full, and item index 0 is always the text shown at the top of the screen area, and so on down. So, in order for this to be a "scroll," we need to shift the contents of the array "up," that is, toward the 0 index. The 0 index element will fall into the "bit bucket." Fortunately, JavaScript arrays have some neat utility functions, among them `shift()`, which does precisely what we need. Also quite convenient is the `push()` method, which adds an element to the end of the array. This is nice because we don't have to worry about replacing a certain index or anything like that; `shift()` reduces the array's size to 10 elements, and `push()` then brings it back up to 11, and the net result is precisely what we want. Once that is done, we generate some simple markup from the contents of the array, and update the `divActivityScroll <div>`. The last piece of the puzzle is to scroll the contents of the `<div>` far enough so that the last element is always shown entirely. The problem here is that some messages returned by the server will actually take up two lines. So, if we didn't do this final scroll, we'd find that some messages get cut off on either the top or bottom, and sometimes the activity scroll text would extend downward a bit; both results are undesirable. Scrolling the whole `<div>` to an arbitrarily large value effectively pushes all the text down as far as it will go so that nothing is cut off, and in doing so, the `<div>` does not need to scroll.

StoreFuncs.js

I want to jump around just a bit here and look at the two functions in this file that relate to when the player is in a store and can purchase items. These two functions are perhaps the simplest examples that show the structure that most of the remaining functions will take. Listing 10-5 shows this code.

Listing 10-5. *The Entire Contents of StoreFuncs.js*

```
/**
 * This function is called when the player steps on a store trigger tile.
 */
function showStore() {
  if (xhr.request == null) {
    sendAJAX(showStore, "store.jsp", "", null);
  } else {
    showSecondaryView(xhr.request.responseText, Globals.VIEW_STORE);
    return true;
  } // End xhr.request == null if.
} // End showStore().


/**
 * This function is called when the player purchases an item in a store.
 */
function purchaseItem(inWhichItem) {
  if (xhr.request == null) {
    sendAJAX(purchaseItem, "purchaseItem.command", "?whichItem=" +
      inWhichItem, null);
  } else {
    updateActivityScroll(xhr.json.mg);
    return true;
  } // End xhr.request == null if.
} // End purchaseItem().
```

Both of these functions are called to fire off an Ajax request. `showStore()` is called when the user steps onto a tile right in front of a store, thereby entering the store. `purchaseItem()`, as one would expect, is called when the player decides on an item to purchase. Note the overall structure of both: a simple `if` checks to see whether `xhr.request` is `null`, and an `else` block. If it is `null`, a call to `sendAJAX()` is made. If it is not `null`, the `else` block kicks in, and the store view is shown (in the case of `showStore()`) or the activity scroll is updated (in the case of `purchaseItem()`). Most of the other functions we'll look at have this same basic structure, so it is important to understand the flow through them. To help with that understanding, Figure 10-8 shows a flow diagram of a call to one of them, and how `sendAJAX()` is involved.

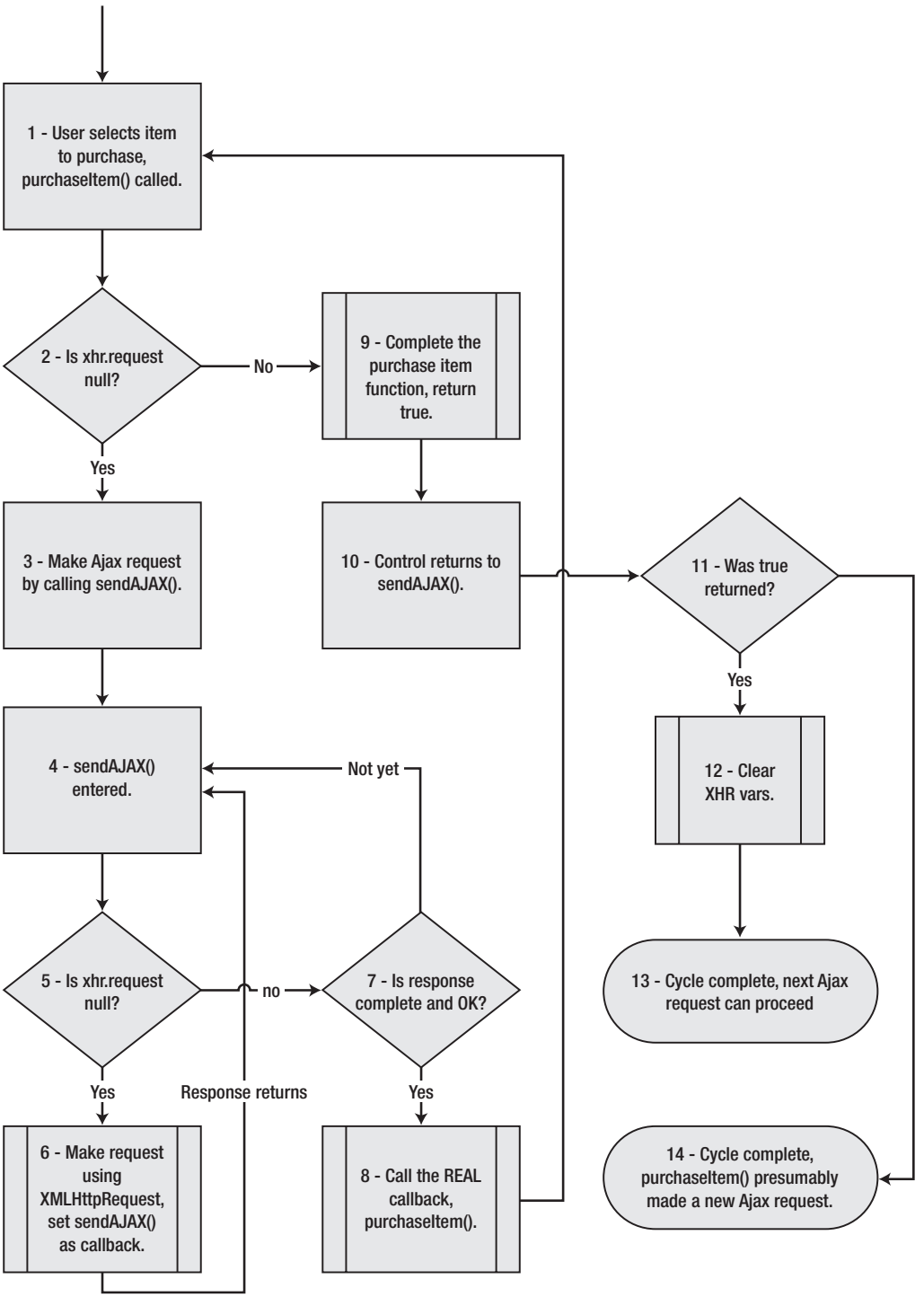

Figure 10-8. *Flow diagram of a typical Ajax request in AJAX Warrior*

KeyHandler.js

The next important piece of code to examine is the key handler code as defined by the function `keyUp()`. This function handles all the key presses in the game (as registered with the browser in the `init()` function).

The first thing we see is our code to get the key code pressed in a cross-browser fashion:

```
var ev = (e) ? e : (window.event) ? window.event : null;
if (ev) {
  keyCodePressed = (ev.charCode) ? ev.charCode:
    ((ev.keyCode) ? ev.keyCode : ((ev.which) ? ev.which : null));
}
```

After that comes an `if` statement with a number of `else if`s in it. The logic branches depending on which view is current, since different keys are valid in different views. For instance, the E key, which enters a community when walking around the map, does not have any meaning while in battle. Within each `if` block is a `switch`, like the following:

```
switch (keyCodePressed) {
  case Globals.KEY_LEFT_ARROW:
    updateActivityScroll("West");
    updateMap("left");
  break;
  case Globals.KEY_RIGHT_ARROW:
    updateActivityScroll("East");
    updateMap("right");
  break;
 … and so on …
}
```

The vast majority of the cases simply call the appropriate function, such as `updateMap()` shown here (the call to `updateActivityScroll()` as well is common). There is some added complexity when buttons are pressed when the spell casting view or weapon switching views are shown; they can be called from either normal or battle view, so the code has to switch back to the appropriate view. Likewise, the cases for Battle mode are a bit more complex; for instance, the arrows keys can indicate movement, or they can indicate the direction of fire if the player previously pressed F for Fire Projectile. But again, by and large, this entire chunk of code amounts to "If key X was pressed, call function X()… if key Y was pressed, call function Y," and so on (where there is a different set of `if`s for each view).

ViewChangeFuncs.js

This file contains a number of functions that switch between the various views in the game. The views are

- Normal view (when the player is walking around the map)
- Battle view (when the player is fighting a character)

- Inventory view (when the player is viewing their item inventory)
- Spell casting view (when the player wants to cast a spell)
- Weapon switching view (when the player wants to switch what weapon they are currently using)
- Help view (when the player is viewing the game help)
- Talking view (when the player is talking to a character)
- Store view (when the player is looking at the items they can purchase in a store)
- Game end view (when the player has either died or won the game)

`showHelp()` is called to show game help. `showMapView()` is called to show the normal map view. `showGameEnd()` is called to show the game end view (passing it the value "won" or "died" to indicate which view to show). `displayInventory()` is called to show the player's inventory.

All of these views make use of the `showSecondaryView()` function. This function accepts two parameters: the first is the markup to display, and the second is which view to display. Using these parameters, it populates the appropriate `<div>` and shows it, and also sets the `currentView` member of `gameState` appropriately.

You might be wondering why there is no `showSpellCasting()` or `showWeaponSwitching()` function, or others that change the view. The reason is that the ones that have functions here are essentially information-only views; no functionality is hidden within them. When you view inventory, you cannot select an item to use, for instance. However, spell casting requires two things: showing the spell casting view, and handling the casting of a spell. Because of this, you'll find a `castSpell()` function elsewhere that does what any of these would do—that is, show the spell casting view—but it also handles when the user picks a spell to cast. We'll see how that works soon.

CastSpell.js

Well, since I went and just brought it up, let's take a quick look at `castSpell()` right now! You can refer to the source file downloaded from the Apress website.

As before, you can see the overall structure repeated in terms of the branching logic based on whether or not `xhr.request` is `null`. When this function is called (which means the user pressed the C key), we need to simply show the spell casting view. In this case, the parameter `inWhichSpell` would not be passed, so the `else` block of the `if (inWhichSpell)` check is executed. This sets `gameState.spellCast` to false, and fires off an Ajax request to retrieve the markup for the spell casting view. When `sendAJAX()` receives the response, it calls back to `castSpell()`. At this point, the `else` block of the `if(xhr.request == null)` check executes. Within it, the `else` block of the `if (gameState.spellCast)` executes. This switches to the spell casting view and also evaluates that `<script>` block we spoke of earlier that lists what spells the player can cast. Now, at this point, you'll recall that the `keyUp()` function will be using a different set of `if` checks. So, when the user now presses a button corresponding to a spell they want to cast, it calls `castSpell()` again, but this time it passes in the code for the spell the player wants to cast. So, the `if (inWhichSpell)` check winds up executing the `if` portion,

which makes an Ajax request to cast the spell. When `sendAJAX()` gets the response back, it again calls back to `castSpell()`. Again, the `else` block of the `if (xhr.request == null)` check executes, but time, because `gameState.spellCast` is true, the `if` portion of the `if (spellCast)` check executes. This updates the player's information and outputs whatever message the server returned to the activity scroll. That is a complete spell casting cycle from start to finish. That is also how a single function, `castSpell()`, can handle all the various flows through it.

Also in this JavaScript file is a function `doesPlayerHaveSpell()`. This is called when a key is pressed in Spell Casting mode to determine if the spell requested is one the player has to cast. It checks against the string `eval()`'d previously to get the value of `gameState.spellsPlayerHas`. This is nothing but a simple array scan to see if the requested code is present.

SwitchWeapon.js

If you just read the section on `CastSpell.js`, then you already more or less know all about `SwitchWeapon.js`! It is virtually identical. Please do have a look, and compare the two so as to be sure I am not pulling your leg.

Conversation.js

This source file contains four functions related to talking to a character: `startTalking()`, `showNodeReplies()`, `stopTalking()`, and `talkReply()`.

`startTalking()` is, quite logically, what is called when the player comes in contact with a character they'll talk to. It is fairly straightforward: it shows the appropriate character image, switches to the talking view, shows the initial thing the character says (the character always talks first; pretend the player said "Hello!" automatically), and finally it shows the initial set of replies the player can make.

`showNodeReplies()` is the function that literally shows those replies, grabbing them from the `xhr.json` field.

`stopTalking()` does precisely what it says: it stops talking by setting the view back to normal (map) view.

Finally, `talkReply()` send an Ajax request when the player selects their reply. Let's have a quick look at this code:

```
function talkReply(inWhichReply) {

  if (xhr.request == null) {

    sendAJAX(talkReply, "talkReply.command", "?reply=" + inWhichReply, null);

  } else {

    if (xhr.json.mo == Globals.MODE_NORMAL) {
      updateMap();
      document.getElementById("imgCharacter").src =
        Globals.imgTransparent.src;
      showMapView();
```

```
    } else {
      showNodeReplies();
    }
    return true;

  } // End xhr.request == null if.

} // End talkReply().
```

OK, so, as usual, we have the typical branching we've seen a number of times now. What is interesting is the `else` block, when the response comes back. If the `mo` member of the `xhr.json` reply indicates we are now in normal mode, that means either the character got spooked by us and ran away, or they gave us a key. In either case, we're going to switch back to normal map view, and call `updateMap()` to get the correct view again. If the response did not indicate a switch back to normal mode, then the conversation is continuing, in which case the server will have returned a new set of replies the player can make. So, we call on `showNodeReplies()` once more to display them.

And that's how we talk to a nonexistent monk, thief, peasant, or guard!

GameFuncs.js

`GameFuncs.js` contains what would be considered the "core" client-side code of AJAX Warrior.

The first function we encounter is `updatePlayerInfo()`:

```
function updatePlayerInfo(inName, inHealth, inGoldPieces, inHitPoints) {

  document.getElementById("divName").innerHTML = inName;
  document.getElementById("divHealth").innerHTML = "Health: " + inHealth;
  document.getElementById("divHitPoints").innerHTML =
    "Hit Points: " + inGoldPieces;
  document.getElementById("divGoldPieces").innerHTML =
    "Gold Pieces: " + inHitPoints;

} // End updatePlayerInfo().
```

Pretty simple: pass if the player's name, health, gold pieces, and hit points, and the appropriate `<div>` elements will be updated.

Next up is `toggleTalkAttack()`:

```
function toggleTalkAttack(inWhichState) {

  if (xhr.request == null) {

    sendAJAX(toggleTalkAttack, "toggleTalkAttack.command", "", null);

  } else {
```

```
    updateActivityScroll(xhr.json.mg);
    // Change the border state.
    if (gameState.talkAttackMode == Globals.PLAYER_TALK_MODE) {
      document.getElementById("imgBorder").src = Globals.imgATTACKING.src;
      gameState.talkAttackMode = Globals.PLAYER_ATTACK_MODE;
    } else {
      document.getElementById("imgBorder").src = Globals.imgTALKING.src
      gameState.talkAttackMode = Globals.PLAYER_TALK_MODE;
    }
    return true;

  }

} // End toggleTalkAttack().
```

Once more, our typical function structure. When the reply returns from the server, we display the message we got back telling us what mode we are now in, and we then toggle the mode as recorded in `gameState`. Finally, we update the border image as appropriate: red for Attack mode, blue for Talk mode.

After that comes `enterCommunity()`. This is a simple call to the server that switches what map we are using from the main Xandor map to one of the community maps (unless the player wasn't standing on a community, in which case the message returned will indicate that). A call is then made to `updateMap()`, passing it `null`, which means it will return the chunk of the map corresponding to where the player begins inside a community.

`pickUpItem()` is next, and it is yet another simple Ajax call to the server. When the response returns, the player's information is updated (because if they picked up gold or a health pack, their info might have changed), and the function also displays whatever message the server returned.

`SaveGame()` is up next:

```
function saveGame() {

  if (xhr.request == null) {

    // Serialize the gameState object.
    var serializedGameState = gameState.serialize();
    sendAJAX(saveGame, "saveGame.command", "", serializedGameState);

  } else {

    updateActivityScroll(xhr.json.mg);
    return true;

  } // End xhr.request == null if.

} // End saveGame().
```

Here, we see a call to `gameState.serialize()`, which returns to us a string of XML representing the pertinent details of the `gameState` instance. An Ajax call is then made, passing this string as POST data (because it is passed as the last parameter; note the third parameter is an empty string—that is the query string). Upon return, we simply display the messages as returned by the server.

Last, we find `updateMap()`. This is quite a large function, but it's also quite simple. First, if `xhr.request` is `null`, we see

```
var queryString = "";

// Null means the player isn't moving.  Happens when the page is first
// shown, and in various other limited situations.
if (inMoveDir != null) {
  queryString = "?moveDirection=" + inMoveDir;
}

sendAJAX(updateMap, "updateMap.command", queryString, null);
```

If a move direction was passed in, we build a query string with that information. This will in essence allow the server to scroll the map, moving our viewport into the map. If it is `null`, that indicates it is the initial request for map data; in that case, the server will return the viewport for the player's starting location on the map.

Once the response comes back, we have

```
// Update the activity scroll, if applicable.
if (xhr.json.dm == "true") {
  updateActivityScroll(xhr.json.mg);
}
```

This code checks whether the `dm` element is `xhr.json` is true, which indicates that there is a message that needs to be displayed. In that case, we pass it to `updateActivityScroll()` to take care of that for us.

Then comes a big segment of code, part of which looks like this:

```
// Update the viewport, if applicable.
if (xhr.json.vu == "true") {
  var i = 0;
  var x;
  var y;
  for (y = 0; y < Globals.VIEWPORT_HEIGHT; y++) {

    for (x = 0; x < Globals.VIEWPORT_WIDTH; x++) {

      var tile = document.getElementById("tile-" + y + "-" + x);

      switch (xhr.json.md.charAt(i)) {
```

```
        case Globals.TILE_BLANK:
          tile.src = Globals.imgTILE_BLANK.src;
        break;
        case Globals.TILE_BRIDGE:
          tile.src = Globals.imgTILE_BRIDGE.src;
        break;
        case Globals.TILE_FOREST_THIN:
          tile.src = Globals.imgTILE_FOREST_THIN.src;
        break;
                    … and so on…

                  }

                }

              }
```

We're going to loop through all the images in the grid (remember all those images we created in `init()`?). For each, we'll examine the corresponding element in the `xhr.json` reply to determine what kind of image should be shown; should it be a mountain, water, a town? Note that this all will happen only if the `vu` member of `xhr.json` is found to be true, which indicates the view has been updated. There's no sense doing all this work if the player has not moved anywhere!

Following this giant `switch` is one more chunk of code:

```
  // Finally, always place the player in the center if not in battle.
  // If in battle, place it where specified.
  if (gameState.currentView == Globals.VIEW_BATTLE) {
    var tile = document.getElementById("tile-" +
      xhr.json.cy + "-" + xhr.json.cx);
      if (gameState.battleEnemyTurn) {
        tile.src = Globals.imgCHARACTER_PLAYER.src;
      } else {
        tile.src = Globals.imgCHARACTER_PLAYER_BATTLE.src;
      }
  } else {
    var tile = document.getElementById("tile-" +
      (Globals.VIEWPORT_HALF_HEIGHT) + "-" +
      (Globals.VIEWPORT_HALF_WIDTH));
    tile.src = Globals.imgCHARACTER_PLAYER.src;
  }

} // End viewUpdated check.

// Update the player info, if applicable.
if (xhr.json.iu == "true") {
  updatePlayerInfo(xhr.json.pn, xhr.json.ht, xhr.json.hp, xhr.json.gp);
}
```

```
// If the last character move resulted in us entering a store, fire
// off the Ajax request to get the markup.  Note the return false... we
// will be starting a new Ajax request, so we don't want to null out the
// applicable variables.  However, before we can start that request, we
// have to null out the variables manually.
if (xhr.json.es == "true") {
  xhr.clearXHRVars();
  showStore();
  return false;
}

// Default return.
return true;
```

First, we need to place the player on the map. Wouldn't be much good without 'em! This will be different depending on whether we are in Battle mode. In this mode, the player can move around the entire viewport, whereas in normal mode they are always simply placed in the middle. Also, in Battle mode, when it is the player's turn, a flashing border appears around the player, and likewise when it is the character's turn. To do this, we need to use a different image depending on whose turn it is since the flashing border is done with nothing but an animated GIF.

After that, we update the player's information, if the `vu` member of `xhr.json` is true.

Finally, if the `es` member of `xhr.json` is true, it means the player is entering a store. In that case, we need to fire off a new Ajax request. So, we call `clearXHRVars()`; otherwise we wouldn't be able to make the new call because `sendAJAX()` would block it (or more precisely, act like it was being called back by `XMLHttpRequest`, which would cause an infinite loop). We then call `showStore()` to make the new call.

BattleFuncs.js

`BattleFuncs.js` contains the functions used during battle with a character. There are three functions: `battleMove()`, `battleEnemyMove()`, and `showProjectile()`.

`battleMove()` is called when the player moves when it is their turn. It is passed the direction the player moved, and an Ajax call is made. When the response is received, the `ph` element of `xhr.json` is checked for. If found, it means that the player fired a projectile weapon (slingshot or crossbow). Note that `ph` indicates whether or not the player hit the character, but here we are only checking whether it is present; whether it hit or not is irrelevant at this point (it will be present either way). If it is found, then a number of fields in `gameState` are updated, including what coordinates the projectile starts at (the player's location), where it stops (either the character or the edge of the viewport), what direction it's traveling in, whether or not it hit the character, and so on. At the end, a timer is started that fires `showProjectile()` every 100 milliseconds.

`ShowProjectile()` is responsible for actually showing an arrow flying (whether fired from the player or a character). It works by updating the coordinates of the arrow with every iteration and overwriting the appropriate tile. It restores the previous tile to what it was before the arrow was on it. When the end is reached, it stops the timer and branches, depending on whose turn it was. If it was the player's turn, it fires off a new Ajax request to allow the character a turn. If it was the character's turn, the player's information is updated. Recall that `sendAJAX()` determines when the player dies, so there is no need to do that check here. Likewise, if we have killed the character, `sendAJAX()` will take care of returning us to normal view as well.

Finally, `battleEnemyMove()` is quite similar to `battleMove()`. The primary difference is that when it completes, it doesn't do anything (like `battleMove()` has to call `battleEnemyMove()`) because it is the player's turn again. Therefore, the code just needs to wait for the player's input.

Note the following:

```
gameState.projectileX = parseInt(xhr.json.p1);
gameState.projectileY = parseInt(xhr.json.p2);
gameState.projectileEndX = parseInt(xhr.json.p3);
gameState.projectileEndY = parseInt(xhr.json.p4);
```

JSON is pretty ignorant when it comes to data types. That is to say, everything is a string! Here, had we simply set the fields to `xhr.json.p1`, for instance, when we tried to increment their values later on, we would have gotten string concatenations instead of math (I know this because I made this mistake; I included comments to prove it!). So, by using `parseInt()`, we get true numeric values instead. Review the discussion in Chapter 2 about data typing in JavaScript if you are unclear about this.

And with that, we have completely looked at the client side of things. Whew, quite a ride! Now we're ready to plunge into the server side of things, and there's at least as much code to look at there, so let's jump right in!

The Server-Side Code

While the server-side code of AJAX Warrior is not all that complex for the most part, there is a fair amount of it. It helps to understand the overall package structure first.

There are a number of packages, all of them under `com.apress.ajaxprojects.ajaxwarrior`. The first is `commands`. AJAX Warrior uses its own simple framework consisting of a front servlet and a number of commands, each corresponding to some specific call the client can make. Those commands are found in the `commands` package.

The `filter` package contains a single servlet filter that performs a check to ensure that a game has been properly started before the request can continue.

The `framework` package contains the classes that make up the application framework, including the front servlet.

The `gameobjects` package contains classes that make up the game itself. These are, for the most part, DTO-type objects.

The `listener` package contains a single context listener used to initialize the game at startup.

There are a few classes in the `ajaxwarrior` package itself, and it is in those classes that we'll begin our exploration.

Globals.java

The `Globals` class is nothing but a holder for a whole bunch of constants used throughout the game code. It includes fields for the tiles that appear in the maps, such as `TILE_BRIDGE` and `TILE_WATER_SHALLOW`. It includes fields for items such as `ITEM_GOLD` and `ITEM_HEALTH`. It contains fields as well for characters, artifacts, and so on. It also includes a number of strings, such as `PLAYER_NO_WALK_TILES`, which contains the codes for the tiles that the player cannot walk on. It includes fields like `TILE_WIDTH` and `TILE_HEIGHT`, which define how big a tile is (32×32 pixels). It also includes fields like `PLAYER_START_HEALTH` and `PLAYER_MAX_HEALTH`, which define characteristics of the player at startup and as the game progresses. There is no executable code in this class to speak of; it is simply a value holder.

Utils.java

You know all those JSON messages we've been talking about? Well, in the `Utils` class you'll find the `writeJSON()` method that creates these messages. Let's have a look at that method now. Please refer to its source as downloaded from the Apress website.

As you can see, it boils down to basically building up the contents of a `StringBuffer` and then outputting that to the response object passed in. The first check performed is to see if we are writing JSON as the result of an exception. If so, the response is simply the `ex` member and the message, which the client will display for us.

After that, a number of checks are performed on the input parameters, for instance to deal with a `null` map chunk, which is the data the client will use to render the current view of the map. Since the view will not always be updated, `null` can be passed here, in which case we need to avoid `NullPointerExceptions`. Likewise, if the view *is* being updated, then the incoming map chunk, which is an `ArrayList`, has to be converted to a `String`, and that is done too.

The JSON message varies depending on what is happening in the game. For instance, when we are in Battle mode, the elements `cx` and `cy` are present; otherwise they are not. Likewise, if a projectile weapon has been fired, then the `p1`, `p2`, `p3`, `p4`, `ph`, and `pd` elements will be present; otherwise they are not. This all keeps the messages as small and efficient as possible by generally not passing superfluous data.

The other method found in this class is `getDescFromCode()`, which is used to get a descriptive string for a given item code. This is used when displaying inventory, for instance. Note that for spells, this method also displays the number of that spell the player has in parentheses after the spell description.

MapHandler.java

The `MapHandler` class is far and away the largest class in all of AJAX Warrior, but it has to be; it has quite a lot of responsibility! In fact, if any class were to be called the heart and soul of this application, `MapHandler` probably stands the best chance of winning that title. Let's jump right in, shall we?

I'll begin by giving you a rundown of the data fields in this class:

- `static Log`: Log instance
- `ArrayList mainItems`: The collection of items placed on the main map
- `ArrayList townAItems`: The collection of items placed on the townA map
- `ArrayList townBItems`: The collection of items placed on the townB map
- `ArrayList villageItems`: The collection of items placed on the village map
- `ArrayList castleItems`: The collection of items placed on the castle map
- `ArrayList mainCharacters`: The collection of characters placed on the main map
- `ArrayList townACharacters`: The collection of characters placed on the townA map
- `ArrayList townBCharacters`: The collection of characters placed on the townB map
- `ArrayList villageCharacters`: The collection of characters placed on the village map
- `ArrayList castleCharacters`: The collection of characters placed on the castle map

- `String currentMapString`: String name of the current map
- `ArrayList currentMap`: Pointer to the map that is currently in use
- `ArrayList previousMap`: Pointer to the previous map that was used before the battle began
- `ArrayList currentItems`: Pointer to the items collection that is currently in use
- `ArrayList currentCharacters`: Pointer to the characters collection that is currently in use
- `ArrayList battleMap`: The battle map currently in use
- `Random generator`: Random number generator

Now we'll start exploring the methods by looking at the constructor. The constructor is a series of calls to `placeItems()`, which is responsible for randomly creating and placing items on the map (such as gold, health packs, and spell scrolls). There is also code specifically to add all the keys and artifacts. This is done for all five maps (four communities and the main map). After that come similar calls to `placeCharacter()`, which randomly places various characters on each map. The key masters are added explicitly in the constructor as well.

Next up we find the ubiquitous `getChunk()` method. This method is called frequently from various commands at various times. Its job is to return an `ArrayList` that represents the viewport of the map. In other words, the collection will contain 13 rows (because that is the height of the viewport as defined in `Globals`) where each row has 13 characters (because that is also the width of the viewport as defined in `Globals`) and each character is one tile on the map that the player sees. This `ArrayList` will be returned as one giant string to the client, which will then use it to render the current view. `getChunk()` begins by calling `getBaseMapChunk()`, which is where the actual map data is retrieved. It then superimposes over that map data the items that are currently on the map with this code:

```
for (Iterator it = currentItems.iterator(); it.hasNext();) {
  GameItem item = (GameItem)it.next();
  x = item.getXLocation();
  y = item.getYLocation();
  // If the next item in the collection is within the viewport, replace
  // the appropriate tile with the appropriate character tile.
  if (x >= inCurrentLocationX &&
    x <= (inCurrentLocationX + (Globals.VIEWPORT_WIDTH - 1)) &&
    y >= inCurrentLocationY &&
    y <= (inCurrentLocationY + (Globals.VIEWPORT_HEIGHT - 1))) {
    int row = y - inCurrentLocationY;
    int col = x - inCurrentLocationX;
    StringBuffer targetRow =
      new StringBuffer((String)chunk.get(row));
    targetRow.replace(col, col + 1, Character.toString(item.getType()));
    chunk.set(row, targetRow.toString());
  }
}
```

As you can see, it is nothing more than iterating over the collection of items, and for each, determining whether it is actually in the player's view. If it is, the appropriate character in the appropriate `ArrayList` element is replaced with the tile code for the item.

The characters are then superimposed onto the map data with code that is virtually identical to that which we have just seen. The end result is that the `ArrayList` that `getChunk()` returns is an accurate view of the map, items, and characters within view of where the player is standing on the map.

The next method we find is `getBaseMapChunk()`, which, as mentioned earlier, returns the chunk of the map for the viewport. It is nothing but a loop that iterates the number of times the viewport is high (13 currently) and retrieves the appropriate `ArrayList` elements and returns them.

After that comes `getCenterTile()`, which is another commonly used function. Its purpose is to retrieve the code of the tile where the player is standing, which is always the center tile in the viewport. This is used many times to determine when the player is standing on items, characters, tiles they cannot actually walk on, and so forth.

Next up we run into `switchMap()`. This method is called, for instance, when entering a community. All of the methods in this class work against the `currentMap` variable, which points to the appropriate `ArrayList` for the map the player is currently on (i.e., Xandor, a town, etc.). When the player enters a community, we need to point to a different `ArrayList`, and `switchMap()` does this. We pass it the name of the map we want to switch to, and it handles all the heavy lifting with code like this:

```
currentMap        = GameMaps.castleMap;
currentItems      = castleItems;
currentCharacters = castleCharacters;
currentMapString  = inWhichMap;
```

An `if…else` check determines which block to execute, and there are five blocks similar to this. `currentMap` is pointed at the appropriate `ArrayList` in the `GameMaps` class, and the items and characters are also updated. We also record the string that was passed into this method for logging purposes.

There is also one special value that can be passed in: "previous". This value is used primarily when battle ends in order to get us back onto the previous map without us knowing what it was. In other words, because the player can enter into battle while walking around Xandor, or while in any of the communities, the code cannot be written to return to any specific map; it can only determine what map to return to at runtime, and it does this based on this "previous" value.

The `placeItems()` method comes next. Passed into this method is the name of the map we're placing items on. Based on that value, a few lines of code execute, such as the following if the value was "main":

```
mainItems         = new ArrayList();
targetCollection  = mainItems;
mapBeingPopulated = GameMaps.mainMap;
numberOfItems     = Globals.NUM_ITEMS_ON_MAIN_MAP;
```

Here, we're first creating a whole new collection of items for this map. We're then pointing the `targetCollection` variable to that collection. This again allows the remainder of the code to be generic and we don't have to know what map we are dealing with. We also need a reference to the map itself that we're adding items to; we'll need to check on it later when we want to determine if the target tile is a valid one for an item to be placed on. Lastly, we get the number of items that should be on this map from `Globals`.

At this point, we begin a loop that iterates the number of times now specified by the `numberOfItems` variable. For each iteration of the loop, we randomly decide what kind of item we're going to place—health, spell scroll, or gold chest—and instantiate a `GameItem` object for it. At this point, we also randomly decide the quantity of each and set it on the object. Next, we randomly pick a tile to place the item on. We call on the `isItemSafeTile()` method, which returns true if it's OK to place an item on the tile, and returns false if not. If we get back false, we choose another location, and keep doing so until we find a safe location. Finally, when we have a location, we add the item to the `targetCollection`, and we are done.

The next method we stumble upon is `placeCharacters()`. This method starts off exactly the same as `placeItems()` in terms of pointing to the correct collection to add to (and first clearing it), pointing to the correct map data, and so forth. It then enters the same kind of loop. However, this method doesn't do any of the actual work! Instead, it makes a call to `createCharacter()`, which returns a `GameCharacter` object. Note that each character is given an ID based on the index of the loop. This is important because this ID is used later when removing a character from the collection, such as when the player kills the character in battle. It then calls `charPickLocation()`, passing it the `GameCharacter` object. These are broken out because they are needed at other times, such as when the player kills a character, so that we can create a new one. Once this is done, the character is added to the collection, and that is that.

The `removeCharacter()` method comes next, and as we mentioned earlier, the ID of the character to remove is passed in. The collection to remove from is determined by examining the `currentMapString` variable, which, as you'll recall, is set by calling `switchMap()`. After that, it is a simple loop to find and remove the character:

```
int i = -1;
for (Iterator it = targetCollection.iterator(); it.hasNext();) {
  i++;
  GameCharacter gc = (GameCharacter)it.next();
  if (inID.equalsIgnoreCase(gc.getId())) {
    // i is now the index of the character to delete.
    break;
  }
}
targetCollection.remove(i);
```

`createCharacter()` is the next method we find. As you'll recall, this is the first one called by `placeCharacters()`. Its job it to randomly determine the characteristics of the new character. First, it instantiates a `GameCharacter` object. Next, it randomly decides what direction the character is going to be moving in. Then, it randomly decides what type of character it is, and sets the applicable characteristics. There is a big `switch` statement in which each case is a specific character type. They are all quite similar, so I'll show just the one for a guard as an example:

```
// Guard.
case 1:
  // 0_thru_2 + 1 = 1_thru_3
  b = generator.nextInt(3) + 1;
  character.setTalkConversation(
    GameConversations.getConversation("guard_" + b));
  character.setImmobile(false);
  character.setType(Globals.CHARACTER_GUARD);
  // Guards will be belligerent about 20% of the time.  0_thru_100
  b = generator.nextInt(101);
  if (b < 20) {
    character.setBelligerent(true);
  }
  character.setGreenKeymaster(false);
  character.setRedKeymaster(false);
  character.setHitPoints(generator.nextInt(26) + 15);
  character.setHealth(generator.nextInt(21) + 80);
  // 0_thru_4
  w = generator.nextInt(5);
  switch (w) {
    case 0:
      character.setWeapon(Globals.WEAPON_DAGGER);
      break;
    case 1:
      character.setWeapon(Globals.WEAPON_STAFF);
      break;
    case 2:
      character.setWeapon(Globals.WEAPON_MACE);
      break;
    case 3:
      character.setWeapon(Globals.WEAPON_SLINGSHOT);
      break;
    case 4:
      character.setWeapon(Globals.WEAPON_CROSSBOW);
      break;
    default:
      log.error("** THIS SHOULD NEVER HAPPEN!");
      break;
  }
  break;
```

The first thing decided is which of the three guard conversations this character will use. Next, we call `setImmobile(false)` to indicate that this character moves around (some do not). Next, we determine whether or not the guard is belligerent (i.e., will always attack the player). We do this by picking a number between 0 and 100. If the value is < 20, then the guard is belligerent. This in effect means that about 20 percent of the time, guards will be belligerent. Well, that would be true if the random number distribution were truly random! It is close enough for our purposes, though; we aren't creating NSA code hashes here after all!

Next, since a guard is not a key master, we pass false to both `setRedKeymaster()` and `setGreenKeymaster()`. After that, we randomly decide on the guard's hit points. Since a guard is likely a pretty strong character, their hit points will always be between 15 and 35. Next we set their health, and again, because guards are quite strong, that value will always be between 20 and 100.

Finally, we randomly decide what kind of weapon the guard has. Guards can use all the weapon types (some characters cannot), so we have a lot of options to randomly choose from.

As I mentioned, this code is similar for all the character types. The ranges of the random values for things like health and hit points are of course different, and some characters, such as serpents, always use a specific weapon (a crossbow in that case), and some characters are always belligerent or never belligerent, so that decision is not present for some. But overall, the basic code structure is the same.

`charPickLocation()` is the next method. Its job is to choose a random location on the map for a character. This code is very similar to that seen in `placeItems()`, and it performs the same check to be sure the tile is valid for a character. Of course, this function uses a different string from `Globals`, since the tiles a character can be on are somewhat different from the tiles an item can be on.

The next method up for bid on The Code Is Right (sorry, couldn't resist!) is `getItem()`:

```
public GameItem getItem(final  int inCurrentLocationX,
  final int inCurrentLocationY) {

  // Calculate the X and Y coordinates the player is standing on.
  int x = inCurrentLocationX + Globals.VIEWPORT_HALF_WIDTH;
  int y = inCurrentLocationY + Globals.VIEWPORT_HALF_HEIGHT;
  // Now iterate over the collection of items for the current map and when
  // we find the item with those X/Y coordinates, return item to caller.
  int i = 0;
  GameItem gi = null;
  for (Iterator it = currentItems.iterator(); it.hasNext();) {
    gi = (GameItem)it.next();
    if (x == gi.getXLocation() && y == gi.getYLocation()) {
      break;
    }
    i++;
  }
  return gi;

} // End getItem().
```

As the name makes clear, it is used to retrieve a `GameItem` object for a specified tile on the map. Specifically, this is used in `PickUpItemCommand` when the player tries to pick up an item. Since we know the player is always in the middle of the viewport, we need to calculate the exact X/Y coordinate for the item because the `inCurrentLocationX` and `inCurrentLocationY` will actually be the upper-left corner of the viewport because that's always what "current location" of the player means outside of battle. So, we simply figure out that X/Y coordinate, then iterate over the collection for the `currentItems` and find the one with those coordinates and return it, or return `null` if it is not found (i.e., the player was not on an item when they pressed P).

`removeItem()` is the next method in line, and it works very similarly to `getItem()` in terms of calculating the X/Y coordinates and iterating over the collection until the item is found. The difference is that when it is found, the loop is exited and the index of the item found is removed.

The next four methods are `isNoWalkTile()`, `isCharacterNoWalkTile()`, `isItemTile()`, and `isItemSafeTile()`. I am grouping them because they are all pretty much the same: they return true or false to indicate whether the specified tile

- Is one that a player cannot walk on
- Is one that a character other than the player cannot walk on
- Currently has an item on it
- Is one we can place an item on

They all work by referencing some constant string in `Globals` (`PLAYER_NO_WALK_TILES`, `CHARACTER_NO_WALK_TILES`, `ITEM_TILES` and `ITEM_SAFE_TILES`, respectively) and seeing if the tile code passed in is found in the string. If so, true is returned; otherwise false is returned. These methods are used in various situations such as placing an item on the map, determining if the player's desired move should be allowed, and so on.

The next method is a pretty big one, `moveCharacters()`. This is called after the player makes a move to move all the characters on the map. First, it begins to iterate over the collection of characters on the map. It gets a reference to the next character in line and checks to see if it is immobile. If it is, then this iteration of the loop is ended with a `continue` keyword.

Next, we get the pertinent details about the character, such as its current location, move direction, and so forth. We now perform a check to determine how far away from the player the character is. If the character is belligerent, and if they are within view of the player, then the character will move toward the player. This is a very basic pursuit algorithm common in many games. One extra thing we do here is we check to see if the character is exactly one space away from the character diagonally. If so, we *do not* move them. This avoids a situation where the character appears to "jump" right on top of the player rather than actually pursue them. This is caused by the fact that if this check was not present, the X *and* Y coordinates would get updated by the subsequent code, which would result in the character now occupying the same tile as the player, causing combat to begin. This would not allow the player the possibility of running away from a character with the character in pursuit, and I felt that was an important thing to allow for.

Once the character's location is updated, we check to see if they wound up on a tile they cannot walk on. If so, we revert to the previous location. This too allows for the player to escape, for instance by walking over a bridge, which characters cannot walk on.

If the character *is not* belligerent or is out of view of the player, we just move it according to its current direction of movement. We first, however, check to see if we have moved a required number of tiles in that direction as defined in `Globals`. If we have, a new direction of travel is chosen.

If the character is still moving, though, we randomly determine whether to move them or not. Thirty percent of the time, a character will not move for this iteration.

A final check is performed to ensure that the character does not walk off the edges of the map. If they get within two tiles of the edge, they revert to their previous location. Note that eventually they'll walk away from the edge; they won't just get stuck there.

When all is said and done, the location of the player is updated (which may not be changed from what it was when this method started), and we update their direction of movement, since that could have changed too.

And that is character movement in a (rather large) nutshell.

The next method seen in this class is `isTileAlreadyOccupied()`. This method returns true if a character already occupies the specified tile. This method is used, for instance, to determine if a character can be placed on a tile when generating the characters for the map.

The next method, `touchingCharacter()`, is used to determine when battle should begin because the player and a character are "touching," that is, occupying the same tile. Here's its code:

```
public GameCharacter touchingCharacter(final int inCurrentLocationX,
  final int inCurrentLocationY, final ArrayList inChunk) {

  GameCharacter gc = null;
  char playerTile = getPlayerTile(inChunk);
  // Calculate the true X/Y coordinate of the player.
  int x = inCurrentLocationX + Globals.VIEWPORT_HALF_WIDTH;
  int y = inCurrentLocationY + Globals.VIEWPORT_HALF_HEIGHT;
  // See if the tile the player is on is a character tile.
  for (int i = 0; i < Globals.CHARACTER_TILES.length(); i++) {
    if (playerTile == Globals.CHARACTER_TILES.charAt(i)) {
      // OK, the tile the player is on is a character.  Now we need to find
      // the character.
      for (Iterator it = currentCharacters.iterator(); it.hasNext();) {
        GameCharacter g = (GameCharacter)it.next();
        if (x == g.getXLocation() && y == g.getYLocation()) {
          gc = g;
        }
      }
      break;
    }
  }
  return gc;

} // End touchingCharacter.
```

As you can see, the applicable `GameCharacter` instance is returned, or `null` is returned if the player is not in contact with a character.

We are almost done! The next method (other than the getter and setter for the `currentMapString` field and the `toString()` method) is `createBattleMap()`. When the player enters into battle, the map they see is constructed by filling the viewport with whatever tile they player is standing on—such as grass or mountains. Here's the code that accomplishes that:

```
public ArrayList createBattleMap(final int inCurrX, final int inCurrY,
  final GameCharacter inCharacter) {
```

```
    ArrayList chunk = getBaseMapChunk(inCurrX, inCurrY);
    char playerTile = getPlayerTile(chunk);
    // Now generate our battle map, which is the size of our viewport, filling
    // it with nothing but the tile the player was standing on (which should
    // always be a ground tile, i.e., dirt, etc.)
    battleMap = new ArrayList();
    for (int y = 0; y < Globals.VIEWPORT_HEIGHT; y++) {
      battleMap.add(StringUtils.repeat(
        Character.toString(playerTile), Globals.VIEWPORT_WIDTH));
    }
    // Now get the battle map, with the character superimposed into it.
    ArrayList bm = getBattleMap(inCharacter);
    previousMap = currentMap;
    currentMap  = battleMap;

    // Return the battle map.
    return bm;

  } // End createBattleMap().
```

As you can see, we get the chunk of the map for the viewport, and then get the tile the player is on. We then construct a chunk to fill the viewport using that tile. We use the handy `repeat()` method of the `StringUtils` class from Commons Lang. This returns to us a string filled with the specified character of length `Globals.VIEWPORT_WIDTH`. We do this `Globals.VIEWPORT_HEIGHT` times, filling an `ArrayList` with each iteration, and return it. That is our battle map!

Finally, we come to the last method in this big class: `getBattleMap()`. This simply returns the battle map that was created by `createBattleMap()`, with one difference: the character is superimposed onto it in its current location, like so:

```
  public ArrayList getBattleMap(final GameCharacter inCharacter) {

    int col = inCharacter.getXLocation();
    int row = inCharacter.getYLocation();
    ArrayList bm = new ArrayList(battleMap);
    StringBuffer targetRow =
      new StringBuffer((String)bm.get(row));
    targetRow.replace(col, col + 1, Character.toString(inCharacter.getType()));
    bm.set(row, targetRow.toString());
    return bm;

  } // End getBattleMap().
```

ClientSideGameState.java

The `ClientSideGameState` object is a simple JavaBean, which is used to store a representation of the client-side `GameStateObject` that we saw earlier. Internally, it has an `ArrayList` for the activity scroll, and two `String` fields, one for `currentWeapon` and another for `talkAttackMode` (and the applicable getters and setters). It also has a `String activityScrollEntry`, but this is used only to populate the instance using Digester when the game is saved, which we'll see later when we discuss the `SaveGameCommand` class.

Aside from the usual JavaBean methods, there is also a `getAsString()` method:

```
public String getAsClientString() {

  // Construct a delimited string where ~~~ is the delimiter sequence.
  StringBuffer sb = new StringBuffer(1024);
  sb.append(StringEscapeUtils.escapeJavaScript(talkAttackMode) + "~~~");
  sb.append(StringEscapeUtils.escapeJavaScript(currentWeapon) + "~~~");
  for (Iterator it = activityScroll.iterator(); it.hasNext();) {
    sb.append(StringEscapeUtils.escapeJavaScript((String)it.next()) + "~~~");
  }
  return sb.toString();

} // End getAsClientString
```

This method is used when a game is being continued. It constructs a delimited string, delimited by three tildes in a row (`~~~`) because that should be a safe delimiter in terms of it never naturally occurring in any of the data. Recall that the `reconstitute()` function of the client-side `GameStateObject` knows how to parse this string and use it to populate the `GameStateObject` instance. Now you know how it gets the string in the first place!

GameCharacter.java

The `GameCharacter` class is a simple JavaBean. It represents a character in the game that the player can interact with. It contains these fields:

- `String id`: The ID of this character.
- `char type`: The type of the character.
- `int xLocation`: The horizontal location of this character.
- `int yLocation`: The vertical location of this character.
- `boolean belligerent`: True if this character is belligerent toward the player, false if not.
- `int health`: The health of this character.
- `int hitPoints`: How many hit points this character has.
- `char weapon`: What weapon this character is holding.
- `boolean greenKeymaster`: True if this character can tell where the green key is, false if not.
- `boolean redKeymaster`: True if this character can tell where the red key is, false if not.

- `int moveCount`: A counter for how many tiles the character has moved in a given direction so far.
- `char moveDir`: The direction the character is currently moving in (`n`, `s`, `e`, `w`).
- `boolean immobile`: This flag is set to true when the character doesn't move.
- `GameConversation talkConversation`: The `GameConversation` this character will use.

There is no executable code other than the getters and setters (well, aside from the usual `toString()` method that I tend to use in DTO-type classes such as this).

GameConversation.java

`GameConversation` is another simple JavaBean, or DTO-type class. This one represents and stores a conversation as read in from one of the conversation XML files we looked at earlier. Internally it contains a `String id` field that the conversation is known by, and a `HashMap` containing all the nodes from the XML file, keyed by node ID. Like `GameCharacter`, there is no executable code in this class beyond getters, setters, and `toString()`.

GameConversations.java

`GameConversations` is a class that contains the collection of `GameConversation` objects in an internal `HashMap`.

When AJAX Warrior starts up, the `loadConversation()` method of this class is called repeatedly, one for each conversation script to be loaded (this call is made from the `ContextListener` class, which is coming up). This method uses Commons Digester to parse the conversation XML file, populate a `GameConversation` object, and add it to the collection. Have a look at the Digester rules in the class; I believe that the comments pretty well spell out how it works.

There is finally a `getConversation()` method, which accepts an ID and returns the appropriate `GameConversation` object.

GameItem.java

`GameItem` is yet another simple JavaBean that represents an item in the game such as a health pack, spell scroll, key, or artifact. The members of this class are

- `char type`: The type of the item
- `int xLocation`: The horizontal location of this item
- `int yLocation`: The vertical location of this item
- `int value`: The value of this item, either the number of scrolls if it's a spell scroll, the amount of health if it's a health pack, or the amount of gold pieces if it's a treasure chest
- `char spellType`: What type of spell this is, if it is a spell scroll

Once again, you'll find no meat here: just getters, setters, and `toString()` as far as actual code goes. (I am not counting a field definition as executable code—obviously it *is*, but I think you know what I mean!)

GameMaps.java

The GameMaps class is very much similar to the GameConversations class. It is a container of map data, although unlike GameConversation, there is no GameMap class because each map is nothing but a string of characters. However, internally, each map is stored as an ArrayList, where each 100 characters make up an element in the list. The reason I did it this way was so that it would be quick and easy to access a given row—no offset math to perform to get the proper substring or anything like that. It also makes expanding the maps very easy (hint, hint!).

Each of the five maps has its own ArrayList: mainMap, townAMap, townBMap, villageMap, and castleMap. The method loadMap() is called for each of these maps from the ContextListener class at startup, and the name of the map to load is passed in. The following code is then used to read the file and populate the appropriate ArrayList:

```
try {
  loader = Thread.currentThread().getContextClassLoader();
  stream = loader.getResourceAsStream(
    "map_" + inWhichMap + ".dat");
  isr    = new InputStreamReader(stream);
  br     = new BufferedReader(isr);
  String line = null;
  mapToLoad.clear();
  int lineCount = 0;
  int charCount = 0;
  while ((line = br.readLine()) != null) {
    lineCount++;
    charCount += line.length();
    mapToLoad.add(line);
  }
  log.info("Map loaded (lines: " + lineCount + ", chars:" +
    charCount + ")");
  if (log.isDebugEnabled()) {
    log.debug("GameMaps." + mapVarName + "=\n" + mapToLoad);
  }
} catch (Exception e) {
  e.printStackTrace();
} finally {
  try {
    br.close();
    isr.close();
    stream.close();
  } catch (Exception e) {
    log.debug("Exception closing: " + e);
  }
}
```

Prior to this, the variable mapToLoad is pointed to the appropriate ArrayList field, so this code works for any of the maps. Note that way the thread's class loader is used to gain access to the files—which explains why they have to be in WEB-INF/classes: they have to be in the classpath for the class loader in order for them to be found.

GameState.java

`GameState` is the class that stores the current state of the game: information such as what map is in use, what inventory the player has, what character they are talking to, and so on. This class is largely just a simple JavaBean with a bunch of fields and the appropriate getters and setters. There is, however, some actual code in it. First, let's see what fields are present:

- `int currentLocationX`: Player's current X location on the current map.
- `int currentLocationY`: Player's current Y location on the current map.
- `boolean inCommunity`: Flag to tell us whether we are in a community.
- `int mainLocationX`: Player's X location on the main map, saved when entering a community.
- `int mainLocationY`: Player's Y location on the main map, saved when entering a community.
- `int mapLocationX`: Player's X location on the current map, saved when entering battle.
- `int mapLocationY`: Player's Y location on the current map, saved when entering battle.
- `String name`: Player's name.
- `int health`: Player's health.
- `int hitPoints`: Player's hit points.
- `int goldPieces`: Player's gold pieces.
- `boolean attackMode`: Flag that tells us whether the player is currently in Attack mode.
- `MapHandler mapHandle`: The `MapHandler` object associated with this game.
- `LinkedHashMap inventory`: A `Map` of all the items the player is currently holding.
- `char currentMode`: The current mode the game is in.
- `boolean playerWon`: A flag that gets set when the player has won.
- `boolean playerDied`: A flag that gets set when the player has died.
- `GameCharacter talkCharacter`: The `GameCharacter` object the player is currently talking to, if any.
- `String talkNode`: What node in the conversation is current, if player is talking to a character.
- `int karma`: The current karma value of the character the player is talking to.
- `GameCharacter battleCharacter`: The character the playing is doing battle with.
- `char currentWeapon`: The weapon the player is currently using. If a blank space, then the player is using their bare hands.
- `boolean freezeTime`: A flag that indicates whether the Freeze Time spell has been cast.

- `int freezeTimeCounter`: A counter specifying how many moves are left for the Freeze Time spell.
- `int winsToHPIncrease`: The number of battle wins the player must get before the next hit point increase.
- `int numBattleWins`: A counter of how many wins the player has had since the last hit point increase.
- `ClientSideGameState clientSideGameState`: The client-side `GameState` object representation.

Aside from the pedestrian getters and setters, we have `addToInventory()`:

```
public void addToInventory(final char inItem, final Object inValue) {

  // If the item being added is a spell, we want to increase the
  // count for that spell by 1, if there are any in inventory already, or just
  // add if fresh if it is not there already.
  if (inItem == Globals.SPELL_FIRE_RAIN ||
    inItem == Globals.SPELL_HEAL_THY_SELF ||
    inItem == Globals.SPELL_FREEZE_TIME) {
    Integer iSpellCount = (Integer)inventory.get(Character.toString(inItem));
    if (iSpellCount == null) {
      iSpellCount = new Integer(0);
    }
    int spellCount = iSpellCount.intValue();
    spellCount += ((Integer)inValue).intValue();
    inventory.put(Character.toString(inItem), new Integer(spellCount));
  } else {
    // The item is NOT a spell, just add it outright.
    inventory.put(Character.toString(inItem), inValue);
  }

} // End addToInventory.
```

For spells, there is an added complication: we need to add some number of spells to the count for that spell if the player already has that spell, or we need to add the spell fresh if they do not have it already. If it is not a spell, it is a straightforward add. Note that in all cases except spells, we have nothing to add—there is no `Integer` value to add; it is just a placeholder. So, in those cases, the caller of this method would have passed a new `Object`, just to have something to add.

There is, as you've probably guessed, a corresponding `removeFromInventory()` method. It works very similarly to this, except in reverse. Again for spells, the count is retrieved and reduced by 1. If that leaves 0, then the spell is removed from the inventory entirely.

Lastly, we have a `checkGameWon()` method. This method simply checks the inventory to see whether the player has all five artifacts. If they do, the method returns true, and the player will be shown the game end screen with the game won graphic.

This object is serialized when a game is saved, so by extension, anything it is composed of must likewise be `serializable`. Most of the classes in this package in fact are, for just this reason.

Lastly, note the `clientSideState` field. This is the string (XML) representation of the `GameStateObject` on the client, sent in when the game is saved. So, by serializing this object, we in effect save the client-side state as well, with no extra effort required.

GameTalkNode.java

Once more, we have a simple JavaBean. This class represents one of the nodes in a conversation (i.e., one row in the Excel spreadsheet). <sarcasm>It contains an overwhelming number of fields: 3!</sarcasm>

- `String id`: The ID of this node
- `String response`: The character's response for this node
- `LinkedHashMap replies = new LinkedHashMap()`: The collection of possible player replies

There is no executable code, other than the getters and setters, in this class.

GameTalkReply.java

The final class in the `gameobjects` package is `GameTalkReply`. It contains the individual pieces of information for a reply that a player can make for a given node. The `replies` member of the `GameTalkNode` is a collection of three `GameTalkReply` objects. That is also why a `LinkedHashMap` is used: I wanted the iteration order to be known, and `LinkedHashMap` provides that (but still gives you the benefit of a `Map`). This class contains these fields:

- `String id`: The ID of this reply; always `1`, `2`, or `3`
- `int karma`: The change in karma this reply will elicit
- `String target`: The node this reply will jump to
- `String replyText`: The text of this reply

There is no executable code, other than the getters and setters, in this class.

■ **Note** One general note about the classes in the `gameobjects` package: All of them, except for `GameMaps` and `GameConversations`, implement `Serializable` because they are saved as part of `GameState` when the player saves their game. These two exceptions do not need to be saved as they are read in when the application starts and are not altered after that. They also contain a rather large amount of data potentially, so we wouldn't want to store them in session, nor would we want to write them to persistent storage. The session object can be rather large by virtue of the fact that a single `GameConversation` object may be stored in it when the player is talking to a character. There is no sense is exacerbating the situation by storing *all* of them and then *all* of the maps!

SessionCheckerFilter.java

This filter is responsible for checking any request that comes into the front servlet and determining whether a game has been properly started. If not, the request is redirected back to the starting page (`index.jsp`). The `doFilter()` method starts by getting the path that was requested. It then checks to see if the `/startGame` command was requested. This is the very first request that would go through the front servlet in the normal flow, and at that point the game wouldn't have been properly started. So, if this exception wasn't present, the game could never be started because this filter would always redirect the request.

If the request was not to the `/startGame` command, we check for the presence of the `Globals.GAME_PROPERLY_STARTED` attribute in session. Only if it is present do we allow the request to continue; otherwise we redirect to `index.jsp`.

The idea is simply to not allow requests to the game that are not within the context of a properly started game.

ContextListener.java

The `ContextListener` class executes when the context starts up and is responsible for performing some basic initialization required before any game can actually start: reading in the maps and the conversation XML files. In fact, `contextInitialized()` is simply a series of calls to `GameMaps.loadMap()`—five in total, and for each call we pass the name of the map to load. After that is a series of calls to `GameConversations.loadConversation`—12 in total, and for each call we pass the name of the conversation to load.

So, all of the *actual* work is done by the `GameMaps` and `GameConversations` classes, which we'll see in a bit. We simply pass them the name of the map or conversation to load, and they take care of the details.

FrontServlet.java

The `FrontServlet` class is the single servlet that makes up, along with the `CommandResult` class in the `com.apress.ajaxprojects.ajaxwarrior.framework` package, the framework AJAX Warrior is built on. It is not quite Struts or JSF, but it does the trick!

`FrontServlet`'s `doGet()` method calls on `doPost()`, where the work is done. This work consists of the following:

1. For the requested URL, everything is stripped off except the command. For example, the form in `index.jsp` submits to `/startGame.command`, so everything is stripped off until we just have `startGame`, which is the requested command.
2. For the requested command, the appropriate `Command` class is instantiated, passing the `HttpServletRequest`, `HttpServletResponse`, and `ServletContext` objects to its constructor.
3. The `exec()` method of the `Command` class is then called, and the result, which is a `CommandResult` object, is captured.
4. The `finish()` method of the `Command` class is called.
5. The `CommandResult` is examined. If it is `null`, the response is assumed to be finished and that is the end of processing. If it is not `null`, `doRedirect()` is called. If it returns true, a redirect to the URI set in the `CommandResult` is sent to the client. If it returns false, a forward is done to the URI.

If an exception occurs at any point in this process, `Utils.writeJSON()` is called so that the exception can be returned to the client for display via an alert pop-up.

As you can see, this is not a full-featured framework. And since all of the commands are hard-coded in it, it is not terribly flexible. However, it *is* more than adequate for our purposes, and allows us a centralized location to do the common tasks required of most tasks, which keeps redundant code to a minimum. If this were a game I was intent on selling on its own, I would certainly begin by externalizing the command mappings, and extending the framework from there. But this will suffice for the purposes of this book.

CommandResult.java

As mentioned earlier, a `CommandResult` object is returned by any `Command`. Again, because this isn't a full-fledged framework, the resource to return to the client is defined by each `Command`, rather than by any centralized location (i.e., declared forwards in Struts, for instance). The `CommandResult` has two members: `path` and `redirect`. The `path` is the URI to forward or redirect to (a forward is a transfer to another resource on the server that doesn't involve the client; a redirect is a response to the client telling it to make a new request for a resource and providing the address for the client to request). `redirect` is true when we want to do a redirect, and false otherwise (which means do a forward). There is a getter and a setter method for both fields, and that's all there is to the class.

As noted in the discussion of `FrontServlet`, a `Command` can return `null` instead of a `CommandResult` instance. In fact, this is what the majority of them do because they'll have called `Utils.writeJSON()`, at which point the response is fully formed.

Command.java

Jumping over to the `com.apress.ajaxprojects.ajaxwarrior.commands` package, we first find the `Command` class. This is a base class that all other `Commands` inherit from. It has a number of protected fields: `HttpServletRequest request`, `HttpServletResponse response`, `HttpSession session`, `ServletContext ServletContext`, `GameState gameState`, and `MapHandler mapHandler`. This means that any class inheriting from `Command` will automatically have access to these members, which makes for very clean code.

Here's the constructor responsible for populating these members:

```
public Command(final HttpServletRequest inRequest,
  final HttpServletResponse inResponse,
  final ServletContext inServletContext) {

  request        = inRequest;
  response       = inResponse;
  session        = request.getSession(true);
  servletContext = inServletContext;
  gameState      = (GameState)session.getAttribute("gameState");
  // When the StartGameCommand is called before a game has started, gameState
  // will be null, so we have to check for that lest we throw an NPE.
  if (gameState != null) {
    mapHandler = gameState.getMapHandler();
  }

} // End constructor().
```

The Command class has two other methods: exec() and finish(). The exec() method implements the behavior of the Command. finish() is called by FrontServlet after exec() has been called. Let's look at finish() now:

```
public void finish() {

  if (gameState != null) {
    gameState.setMapHandler(mapHandler);
    session.setAttribute("gameState", gameState);
  }
  log.debug("Command done");

} // End finish().
```

Obviously there isn't much to it, but this method's role is vital. Many of the Commands will alter various elements in GameState. This object is stored in session, and must be updated when each Command completes. finish() does this, and also updates mapHandler in gameState. The MapHandler instance is also updated by most Commands, and so must be updated in gameState as well.

BattleMoveCommand.java

We have seen a couple of relatively small and simple commands, but now it is time to look at a larger and more complex one. This is the command that is called when the player makes a move during battle.

First we get a reference to the GameCharacter object that the player is battling from gameState. Next, we record the player's current location, as well as the location of the character. Then, we get the direction in which the player moved from the request. If that direction begins with the string "projectile_", it means that the player has fired a slingshot or crossbow. In that case, we begin to populate a HashMap that will contain all the details of the projectile firing we'll need, beginning with the location where the projectile starts, which is the same as the player's location. Both locations are put into the HashMap under the keys p1 and p2, respectively.

Next, we find a giant switch statement with eight cases. The first four are quite simple: whichever direction the player moved in, assuming they did not fire a projectile, we update the player's location as appropriate.

If the player fired a projectile, however, we determine whether or not the character is hit. This amounts to checking that either the X or Y coordinate matches that of the player, depending on which direction the player fired in. For instance, to determine if the player hit the character while firing up, we do this:

```
if (currentLocationX == gcXLocation && currentLocationY > gcYLocation) {
  log.info("Projectile hit (up)");
  projectileHitEnemy = true;
  projectileInfo.put("ph", "true");
  projectileInfo.put("p3", Integer.toString(gcXLocation));
  projectileInfo.put("p4", Integer.toString(gcYLocation));
} else {
  message = "You missed!";
```

```
    projectileInfo.put("p3", Integer.toString(currentLocationX));
    projectileInfo.put("p4", Integer.toString(0));
  }
```

Obviously, in this case, the X coordinates must be the same, and the Y location of the player must be greater than the character's Y location. If both conditions are true, we put an element in the `HashMap` named `ph` set to true, and we set the ending coordinates of the projectile's travel to the coordinates of the character. If the projectile misses, we set an appropriate message to display to the user, and set the ending coordinates of the projectile's travel to the edge of the viewport.

After doing all that, we check to see if the projectile hit the character by checking the flag variable `projectileHitEnemy`. If it's true, we call the `calculateDamage()` function to determine how much hurt we put on our enemy.

The underlying calculation performed takes the `DAMAGE_BASIS` value (which is some value less than 1) and multiplies that by the amount of damage defined for the current weapon in use, or bare hands. Next, the code divides the hit points of the player by the `HIT_POINT_DIVIDER` value, and multiplies that outcome by the current value of damage. Finally, the code checks to make sure at least 1 unit of damage is done, and also that the amount of damage is not greater than the value of `COMBAT_MAX_DAMAGE`. Finally, this final damage value is returned.

Next, the code performs a check to see whether the player ran away from battle by moving off an edge of the viewport. If so, the mode is switched back to normal and the player is called an unworthy coward (I never said it was a kind game!).

Next, the code sees if the player and the character now occupy the same space. If so, the first thing done is to move the player back to where they were before this move. Next, the `calculateDamage()` method is again used because now the player has attacked the character. The code then checks to see if the enemy was vanquished. If so, one last check is performed: was this one of the key masters, and if so, does the player have their key already? If it was a key master and the player does not already have their key, tell the foolish human the quest is over because there is no longer any way to win the game! Finally, if the player did not kill the character, the code reports back to the player how much damage they did.

After this block of code, we have a bit more work to do. First, we set the player's coordinates to the updated values. These values may or may not be different from before. They'll be different unless the player attacked the character.

This block of code is last:

```
// See if the player defeated the enemy.
if (battleEnds) {
  HashMap vals = endBattle(gameState, mapHandler, gc, ranAway);
  // Update the message to return as long as we didn't run away.
  if (!ranAway) {
    message = (String)vals.get("message");
  }
  mapData = (ArrayList)vals.get("chunk");
} else {
  // Get the map data we'll return, still showing battle mode.
  mapData = mapHandler.getBattleMap(gc);
}
```

Regardless of how the battle ended, if it did end, we call the `endBattle()` method. This method performs a number of common functions always needed when battle ends. First, it restores the previous map and the player's location on it. Second, if the player did not run away, which would mean they won, then we calculate randomly how much gold they got, and we also figure out if it is time to "level up"—that is, increase their hit points. Initially, it takes five battle victories to increase your hit points. After that, it takes 10 victories, then 15, then 20, and so on. Third, the `removeCharacter()` method of `MapHandler` is called to get rid of the character. Fourth, `createCharacter()` and `charPickLocation()` of the `MapHandler` class are called, in that order, to generate and place a new character randomly. Lastly, the appropriate map chunk is retrieved and returned.

Note that this `endBattle()` method, as well as `calculateDamage()`, are broken out as separate methods because they are used from the `BattleEnemyMoveCommand` that we'll look at next.

Finally, once `endBattle()` completes, we generate our JSON response as usual and this command is finished.

BattleEnemyMoveCommand.java

If you read the description of `BattleMoveCommand`, then you essentially already know all about `BattleEnemyMoveCommand`. This command, as the name implies, is called after the player moves when it is the character's turn.

One difference is a block of code you'll see almost immediately in `exec()`:

```
// If time is frozen, get outta Dodge quickly.
if (gameState.isTimeFrozen()) {
  log.debug("Freeze Time spell in effect");
  int ftc = gameState.getFreezeTimeCounter();
  ftc--;
  if (ftc <= 0) {
    log.debug("Freeze Time spell ended");
    gameState.setFreezeTime(false);
  } else {
    gameState.setFreezeTimeCounter(ftc);
  }
  Utils.writeJSON(false, null, null, false, gameState, false, response,
    false, false, null);
  return null;
}
```

If the player previously cast a Freeze Time spell, then the enemy cannot move now, and this code takes care of that. The spell is in effect for 15 moves, meaning the player can move 15 times before the character can move again.

After that, the code is effectively the same as in `BattleMoveCommand`, except obviously checks are performed to see whether the *character* fired a projectile weapon, whether or not the *player* was hit, whether or not the *player* died, and so on. Characters cannot cast spells, so it is a little simpler by comparison.

As noted, this command makes use of the `calculateDamage()` and `endBattle()` methods in `BattleMoveCommand`, so some redundancy is cut out there.

CastSpellCommand.java

This command is called when the player wants to cast a spell, whether it is during battle or just walking around.

The first step it performs is to get the code of the spell the player selected to cast from the request. Next, a simple rejection is performed:

```
// First, do any simple rejections based on mode.
if (gameState.getCurrentMode() == Globals.MODE_NORMAL) {
  if (whichSpell == Globals.SPELL_FIRE_RAIN) {
    message = "You can only cast that spell in battle";
    canCastSpell = false;
  }
}
```

This code handles the case of the player trying to cast a Fire Rain spell while not in battle, which cannot be done.

Next, assuming `canCastSpell` is true (as it would be unless the rejection fired) is to switch on the type of spell. For Heal Thy Self, we simply set the player's health to 100, and subtract 1 from the inventory count for that spell. For Freeze Time, we call `gameState.setFreezeTime(true)`, which will indicate to other commands that time is frozen. Of course, we again remove it from inventory.

If the spell is Fire Rain, we call `gameState.getBattleCharacter()` to get the `GameCharacter` we are doing battle with. We then calculate how much damage the spell does by multiplying the player's hit points by 2, and then limiting it to 20 points, and subtracting that amount from the character's health. Next, we determine whether or not the player killed the character like so:

```
if (gcHealth <= 0) {
  HashMap vals = BattleMoveCommand.endBattle(gameState, mapHandler,
    gc, false);
  mapData = (ArrayList)vals.get("chunk");
  message = (String)vals.get("message");
  viewUpdated = true;
} else {
  gc.setHealth(gcHealth);
  message = "You cast Fire Rain.  You did " + damage +
    " points damage (" + gcHealth + " remaining)";
}
```

If the player did kill the character, we call the `BattleMoveCommand.endBattle()` method, which returns to us the map chunk the player should now see (which would be from the map they were on when they entered into battle), as well as the message (which tells them how much gold they won, and whether they increased their hit points). If the character still survives, we send back a message stating how much damage was done and how much health the character has left, and set the new health value for the character.

DisplayInventoryCommand.java

This is a very simple command that does two things. First, it puts the current `gameState` instance in the request as an attribute under the key `gameState` (I was not feeling particularly

creative that day!) and then forwards to `displayInventory.jsp`. The response is rendered and returned to the client, which displays the new view with the returned markup.

EndConversationCommand.java

As the name implies, this command is called when the player presses E during conversation to end talking right away. Its `exec()` method is pretty straightforward:

```
public CommandResult exec() throws Exception {

  log.debug("EndConversationCommand.exec()...");

  // Put us back in normal mode.  Be sure the character moves off the
  // player.
  gameState.setCurrentMode(Globals.MODE_NORMAL);
  GameCharacter gc = gameState.getTalkCharacter();
  mapHandler.charPickLocation(mapHandler.getCurrentMapString(), gc);
  gameState.setTalkNode(null);
  gameState.setTalkCharacter(null);

  // Get the chunk of the map data corresponding to the player's new
  // viewport on the map.
  ArrayList chunk = mapHandler.getChunk(gameState.getCurrentLocationX(),
    gameState.getCurrentLocationY());

  // Create our JSON string with the pertinent information and write it out
  // to the response.
  Utils.writeJSON(false, "You abruptly end the conversation.  Shame on thee!",
    chunk, true, gameState, false, response, false, false, null);

  log.debug("EndConversationCommand.exec() done");

  return null;

} // End exec().
```

There isn't too much going on. The mode is reset in `gameState` to indicate the player is no longer talking. Then, the character the player was talking to is randomly placed somewhere on the map. This is done so that the character is not right next to the player when the map is shown again, and a conversation cannot be immediately struck up. Finally, the current talk node and character are cleared in `gameState`, since there is no point in them hanging around. Lastly, our JSON response is written telling the player they quit the conversation; the response also includes the map chunk that will be displayed. All done!

EnterCommunityCommand.java

When the player steps onto a town, castle, or village on the map, they can enter that community, and this command is called on to perform that function.

First, the player's current coordinates on the map are saved in `gameState`. This is done so that upon exiting the community, we can put them right back where they were. Next, the map chunk representing the current viewport is gotten from `mapHandler`, and then `getPlayerTile()` is used to get the code of the tile the player is standing on.

At that point, it comes down to a big `switch` statement where each case is a specific community. As an example, here's what happens when the player is standing on the village:

```
message = "You have entered the unnamed village";
viewUpdated = true;
gameState.setInCommunity(true);
currentLocationX = Globals.COMMUNITY_STARTING_X;
currentLocationY = Globals.COMMUNITY_STARTING_Y;
gameState.setCurrentLocationX(currentLocationX);
gameState.setCurrentLocationY(currentLocationY);
mapHandler.switchMap("village");
chunk = mapHandler.getChunk(currentLocationX, currentLocationY);
```

We first set the message that the player will see in the activity scroll, and we set the flag in `gameState` that indicates the player is in a community by calling `setInCommunity()` on `GameState`. Next, we change the player's coordinates on the map to the starting coordinates as defined in `Globals`. Next, we call `mapHandler.switchMap()`, passing it the value "village". This switches the current map that `mapHandler` will work with to the map for the village. Finally, we get the initial chunk of the map for the community by calling `mapHandler.getChunk()`.

After that, we simply call `Utils.writeJSON()`, and we're done. Of course, if the code found that the player was not standing on a community at all, a suitable message is returned informing them that there is nothing to enter.

PickUpItemCommand.java

When a player tries to pick up an item, this command is called.

The first thing it does is retrieve the chunk of the map that is the current viewport. It then retrieves the tile the player is standing on by using this line of code:

```
char centerTile = mapHandler.getPlayerTile(chunk);
```

This is a simple method that always returns the center tile of the viewport, which is always the tile the player is standing on (when not in battle, which is the only time a player can pick up an item). A call to `mapHandler.isItemTile()` is then made to determine whether or not the tile the player is standing on is an item they can pick up. If not, a message saying there is nothing to pick up is returned.

At this point, if it *was* an item they can pick up, the following line of code is hit:

```
GameItem item = mapHandler.getItem(currentLocationX, currentLocationY);
```

This line gets the appropriate `GameItem` object for the item being picked up, which is then added to the player's inventory. If the item is gold, a check is done to see whether the player is at maximum gold capacity, and an appropriate message is returned. If it is a health pack, a check is done to see whether the player is at maximum health, and an appropriate message is returned. If the item is a spell, the appropriate number of that type of spell is added to inventory. Keys and artifacts are simply added to inventory.

Assuming the item was picked up, the flag variable `removeItemFromMap` is set to true. Next, this check is performed:

```
if (removeItemFromMap) {
  mapHandler.removeItem(gameState.getCurrentLocationX(),
    gameState.getCurrentLocationY());
}
```

As the name says, the `removeItem()` method of `mapHandler` removes the item from the collection of items for the current map. However, it also adds a new item randomly to the map (not necessarily the same type of item that was just picked up).

Lastly, a call to `gameState.checkGameWon()` is made. If the player now has all five artifacts, the response indicates that the player has won, and the appropriate game end screen is shown.

PurchaseItemCommand.java

This command is executed when the player chooses an item in a store to purchase. First, the code of the item they wish to purchase is grabbed from the request. Then, the cost of that item is retrieved from the `Globals` class. Next, if the player has enough gold to purchase the item, a message is constructed saying what item they purchased, and the item is added to their inventory. If they do not have enough gold, then a message is returned indicating that.

Finally, in either case, we call `Utils.writeJSON()` with the message, and we are done.

SaveGameCommand.java

This command is executed when the player requests that their game be saved. It is a fairly simple piece of code.

First, the serialized version of the client-side `GameStateObject` is retrieved from the request using the `RequestHelpers.getBodyContent()` method from Java Web Parts (remember, the XML was POST'd to this command). Then, we use Commons Digester to parse the XML. The result is that we get a `ClientSideGameState` object populated with the activity scroll history and both `talkAttackMode` and `currentWeapon` populated. This object is then added to `GameState`.

As this point, we use an `ObjectOutputStream`'s `writeObject()` method to write out the serialized `GameState` object. Finally, we write a simple JSON response saying the game was saved. Nothing to it! (You really have to love Java's support for serialization!)

ShowCastSpellCommand.java

This command is virtually identical to `DisplayInventoryCommand`, except that it forwards to `weaponSwitching.jsp` instead. 'Nuf said!

ShowSwitchWeaponCommand.java

This command, like `ShowCastSpellCommand`, is the same as `DisplayInventoryCommand`, except that the forward is to `weaponSwitching.jsp` this time around.

StartGameCommand.java

This command is called when the user clicks either the New Game or Continue Game button on the title screen. There are basically two logical flows through this. The first is for starting a new game.

First, the command captures the name that the player entered and uses it:

```
fis = new FileInputStream(servletContext.getRealPath("/WEB-INF") +
  "/gameSaves/" + fileName + ".sav");
request.setAttribute("Error", "A game with the name you " +
  "entered already exists.\\n\\nPlease select a new name and " +
  "try again.  Sorry!");
result = new CommandResult("index.jsp");
```

The expectation here is that the file will *not* be found, meaning that a game with this name has not already been started. The name the player enters is lowercased, and spaces are converted to underscores; that becomes the save filename. If a file with that name is found in `WEB-INF/gameSaves`, then we send the player back to `index.jsp`.

However, if a file with that name is *not* found, then a `FileNotFoundException` is thrown. In that case, the following code executes:

```
gameState  = new GameState();
mapHandler = new MapHandler();
gameState.setCurrentMode(Globals.MODE_NORMAL);
gameState.setCurrentLocationX(Globals.PLAYER_START_X);
gameState.setCurrentLocationY(Globals.PLAYER_START_Y);
gameState.setName(playerName);
gameState.setHealth(Globals.PLAYER_START_HEALTH);
gameState.setHitPoints(Globals.PLAYER_START_HIT_POINTS);
gameState.setGoldPieces(Globals.PLAYER_START_GOLD_PIECES);
gameState.setInventory(new LinkedHashMap());
gameState.setMapHandler(mapHandler);
gameState.setCurrentWeapon(Globals.WEAPON_NONE);
gameState.setWinsToHPIncrease(Globals.HIT_POINT_INCREASE_INCREMENT);
log.debug("StartGame.exec() done (NEW game starting)");
session.setAttribute(Globals.GAME_PROPERLY_STARTED, "true");
result = new CommandResult("main.jsp");
```

This is the sum total of what is required to begin a new game. Not really very much!

After this block of code you'll find the code that executes when a game is being continued. The same kind of file existence check is again performed. If the file is found, we use the `readObject()` method of the `ObjectInputStream` class to get our serialized `GameState` object back as a real object. A flag is also set to indicate that the file was found and the object restored.

Shortly thereafter, that flag is checked. If true, the `GAME_PROPERLY_STARTED` session attribute is set, and the request is forwarded to `main.jsp`, where the game starts up. If the flag was false, we return a message to the user indicating that the file was not found. This is done by sticking the message in a request as an attribute under the key `Error`. `index.jsp` looks for this attribute and renders the appropriate JavaScript `alert()` code to display it to the user.

SwitchWeaponCommand.java

This command is called when the user wants to switch weapons. It is the command called to switch weapons, not just to show the weapon switching view. Its exec() method is

```
public CommandResult exec() throws Exception {

  log.debug("SwitchWeaponCommand.exec()...");

  // Retreive which weapon the player is switching to.
  String paramWhichWeapon = request.getParameter("whichWeapon");
  char whichWeapon = paramWhichWeapon.charAt(0);
  // Set it as current.
  gameState.setCurrentWeapon(whichWeapon);

  // Report back to the user.
  String message = "You are now using your " +
    Utils.getDescFromCode(gameState.getCurrentWeapon(), gameState);

  // Create our JSON string with the pertinent information and write it out
  // to the response.
  Utils.writeJSON(false, message, null, false, gameState, false, response,
    false, false, null);

  log.debug("SwitchWeaponCommand.exec() done");

  return null;

} // End exec().
```

We get the code of the weapon the player wants to switch to, and set the new weapon in gameState. Finally, we write out a JSON response indicating what weapon they are now using.

TalkReplyCommand.java

This command handles the situation where the user is talking to a character and selects one of the three possible replies. The first thing this command does is get a reference to the appropriate GameReply object that corresponds with what the user selected.

Next, it adjusts the karma of the character according to the value of the reply. Then, it checks the karma to see if it is equal to or less than zero. If it is, the following code executes:

```
// Yep, karma at or below 0, run away, run away!!
gameState.setCurrentMode(Globals.MODE_NORMAL);
// Move character so we aren't standing on them.
mapHandler.charPickLocation(mapHandler.getCurrentMapString(), gc);
gameState.setTalkNode(null);
gameState.setTalkCharacter(null);
charResponse = "Argh!  Get away now!";
```

```
// Get the chunk of the map data corresponding to the player's new
// viewport on the map.
chunk = mapHandler.getChunk(gameState.getCurrentLocationX(),
  gameState.getCurrentLocationY());
viewUpdated = true;
```

Here we see that the game state is changed back to normal, which will cause the client side of things to exit Talk mode. Next, we randomly relocate the character so they are not near the player when the view is switched back to the map. We also clear out the current node and character in `gameState`. Next, we send a message to the client saying the character ran away, and finally, we get the viewport map chunk that will be rendered on the client. When all is said and done, this will be used in rendering the JSON response.

Another possible outcome of the karma adjustment is that the character's karma is now equal to or greater than 15. In that case, *if* the character is one of the key masters, the player is given the key. The code for that looks like this:

```
// The character is the green key master and their karma is high enough, so
// now they'll give the player the key.
gameState.setCurrentMode(Globals.MODE_NORMAL);
charResponse = "I place my hope in thee, here is the Green key!";
gameState.addToInventory(Globals.ITEM_KEY_GREEN, new Object());
// Make them no longer a key master, so they don't give us the key again.
gc.setGreenKeymaster(false);
// Move character so we aren't standing on them.
mapHandler.charPickLocation(mapHandler.getCurrentMapString(), gc);
gameState.setTalkNode(null);
gameState.setTalkCharacter(null);
// Get the chunk of the map data corresponding to the player's new
// viewport on the map.
chunk = mapHandler.getChunk(gameState.getCurrentLocationX(),
  gameState.getCurrentLocationY());
viewUpdated = true;
```

The code that deals with the red key master is identical to this, except we replace `green` with `red`. We again set the mode to normal, and then add the appropriate key to the player's inventory. We then change the character so that is no longer a key master; that way, if the player happens to talk to that character again, the player won't be given the key again. Again, the character is relocated, and the applicable variables in `gameState` are nulled. The map chunk is retrieved and is ready for inclusion in the JSON response.

There's another possible outcome of this command: if this is simply another node in the conversation—that is, the character does not run away, and no key is given. In that case, we simply get the target of the reply, and set its ID in `gameState` as the current talk node. Finally, we get the character's response from the target node and return that as the message in the JSON response.

In a nutshell, that's how talking to characters works, as far as the server side of the house goes.

ToggleTalkAttackCommand.java

This command is called to toggle the user between Talk and Attack mode. The code in `exec()` is

```
public CommandResult exec() throws Exception {

  log.debug("ToggleTalkAttackCommand.exec()...");

  // Simply flip the field and return a message stating the new mode.
  gameState.setAttackMode(!gameState.getAttackMode());
  String message = null;
  if (gameState.getAttackMode()) {
    message = "Attack Mode";
  } else {
    message = "Talk Mode";
  }

  // Create our JSON string with the pertinent information and write it out
  // to the response.
  Utils.writeJSON(false, message, null, false, gameState, false, response,
    false, false, null);

  log.debug("ToggleTalkAttackCommand.exec() done");

  return null;

} // End exec().
```

Nothing special is going on here. We call `setAttackMode()` on `gameState`, passing it the negation of its current value. Then we write out the JSON response indicating the mode the player is now in. Short and sweet!

UpdateMapCommand.java

Now, here, near the end, we come to the single biggest command in AJAX Warrior. The `UpdateMapCommand` is responsible for handling the redrawing of the current map display when the player moves. However, it does quite a bit more than that. Let's walk through it, shall we?

First we determine in which direction the player moved by retrieving the `moveDirection` request parameter. If this parameter is not present, it means it is an initial view of the map—that is, when the game first starts, or when we enter a community. (Yes, this command handles walking around Xandor as well as walking around any community. It is all the same as far as the code goes; we just use a different map data set.) Also, if `moveDirection` is `null`, we want to be sure to set the flag in the JSON response to indicate the player information should be updated so that when the game is first started, that information is displayed.

Next, based on the direction of movement requested, we update the player's map coordinates. Immediately after that, we do some bounds checking to be sure they do not scroll off an edge of the map.

Note that the coordinates of the player are a little misleading. The `currentLocationX` and `currentLocationY` members of `GameState`, which convey this information, really are the coordinates of the upper-left corner of the viewport. So, when the coordinates are 0 and 0, then the player will see the upper-left corner of the map, and won't be able to move up or left any further.

The next check only applies if the player is currently inside a community. If they are, and if they are now on one of the exit tiles—either the tile they started out on or the tile directly above or below it—then the community will be exited. To accomplish this, the following code executes:

```
gameState.setInCommunity(false);
gameState.restoreMainLocation();
mapHandler.switchMap("main");
// Get the chunk of the main map after the location is restored.
ArrayList chunk = mapHandler.getChunk(gameState.getCurrentLocationX(),
  gameState.getCurrentLocationY());
Utils.writeJSON(false, "Exited community", chunk, true, gameState,
  false, response, true, false, null);
log.debug("UpdateMap.exec() (exited community) done");
return null;
```

I suspect this code is quite self-explanatory at this point!

Now, after this code, we again get the chunk of the map representing the viewport. We do this because at this point, the player's coordinates could have changed, and the remainder of the checks we need to do have to be on the *new* coordinates.

The next check performed is this one:

```
char centerTile =  mapHandler.getPlayerTile(chunk);
if (mapHandler.isNoWalkTile(centerTile)) {
  log.debug("Is no walk tile, restoring previous");
  message = "Can't move there!";
  currentLocationX = previousLocationX;
  currentLocationY = previousLocationY;
}
```

If the new tile the player would be standing on is one they cannot stand on, like water for instance, then we need to restore their previous location and return a message saying they cannot walk there.

The next set of checks reveals whether the player is walking on one of the special "hidden" door tiles—the tile that must be passed through in order to get to one of the artifacts and that requires a specific key to get through. If the player does not have the appropriate key, the previous location is restored, and the player is hurt a bit. Five of these checks are performed, one for each hidden door. Here's what one of them looks like:

```
if (centerTile == Globals.TILE_WALL_HIDDEN_RED &&
  gameState.getInventory().get(
    Character.toString(Globals.ITEM_KEY_RED)) == null) {
  gameState.setHealth(gameState.getHealth() - 10);
```

```
        message = "You do not have the Red key!  Evil magic attacks you!";
        currentLocationX = previousLocationX;
        currentLocationY = previousLocationY;
        playerInfoUpdated = true;
        if (gameState.getHealth() <= 0) {
          gameState.setPlayerDied(true);
        }
      }
```

Note that we have to check to see whether the player died, and set the flag in `gameState` if so, in order for the appropriate game end view to be shown.

At this point we once again get the map chunk because the coordinates could be different now. Once we do that, we check to see whether the player is now standing on a swamp tile. If so, we reduce their health by 1 and again restore the previous coordinates.

After that, we store whatever the current coordinate values are in `gameState`. These final values take into account all the previous possible changes and resets to the previous coordinates.

Almost done now! If the player did in fact move, and if the Freeze Time spell is not in effect, then we call on the `moveCharacters()` method of `mapHandler` to move all the characters on this map.

Finally, if the Freeze Time spell is in effect, we count down how many moves remain before the spell runs out, and reset things if it does run out.

That is the end of the conditional portions of this command. What remains is what will always happen, beginning with a final get of the map chunk.

Next, we check to see whether the player is touching a character by calling the `touchingCharacter()` method of `mapHandler`. If this method returns true, we determine whether the character is belligerent, and if the player is in Attack mode or Talk mode. If the character is belligerent, or if the player is in Attack mode, then battle begins. If the character is nonbelligerent and the player is in Talk mode, then a conversation begins.

Only one more possibility to deal with! If we stepped on a store tile, and as long as we did not enter into battle (which could happen if a character happened to be standing on a store trigger tile), then we switch to the Store mode.

And of course, at the end, our JSON response is constructed and returned.

Whew! Take a breath! We have now completely explored AJAX Warrior!

Suggested Exercises

Ah yes, suggestions for a game… the best thing about game programming is there are absolutely no limits! You can carry things just as far as your imagination will allow. There are no stuffy corporate rules about how things have to be done, no predefined conceptions of what reality is. Oh, to be sure, the big game houses approach their development efforts about as seriously and professionally as any Fortune 500 business does theirs. But games have always held a special place in programming. Smaller companies, some run out of people's garages early on, can produce something that a great many people will love, and can do so based on nothing but their imagination and their own desire to produce something fun.

So, what kinds of exercises might be worthwhile for you to do with AJAX Warrior? Here is just a small list of suggestions:

- More spells! If you have ever played Dungeons and Dragons, you know that there are more spells that a player can cast than you can shake a magic staff at. Have fun adding as many as you can think of. Because some may require more than simply updating statistics, for instance, you might actually want to show a spell or two being cast and its effect; this would most definitely put your skills to the test.
- Expand the world of Xandor. I purposely tried to design everything to be expandable, which explains why I used the `Globals` class so often. Make the maps bigger, and add some other communities. You could also add more diverse tiles, more characters, more items to retrieve, and so forth.
- Enhance the gameplay with puzzles. Perhaps in addition to getting the appropriate key, the player can solve some sort of puzzle to get through the hidden doors.
- Include a tunnel system. This is something I would have loved to put in this game but simply did not have enough time to do so. The ability to go into tunnels from the mountains, and showing it with a very simple 3-D engine, is doable and would be very interesting.
- Include a boss battle. A final battle with Mallizant would be great. I thought it was a bit of a cop-out to not do it myself, but there are certain time pressures in writing a book! Also, think of how *Lord of the Rings* ended: Sauron is tied to the ring, so once it was destroyed in Mount Doom, he died immediately. I figure Mallizant is the same way: he's tied to the five artifacts he stole. At least, that's my story, and I'm sticking to it!

Summary

Now *that* was one big chapter! I hope you feel, as I do though, that it was well worth the effort. In this chapter we touched on a number of things: JSON, associative arrays, a new way to structure Ajax functions, a new server-side framework, and even a bit of basic game theory. We saw a fair bit of CSS and DOM scripting techniques, and picked up some new JavaScript tricks as well. Most important, we have seen just how fun game programming can be!