

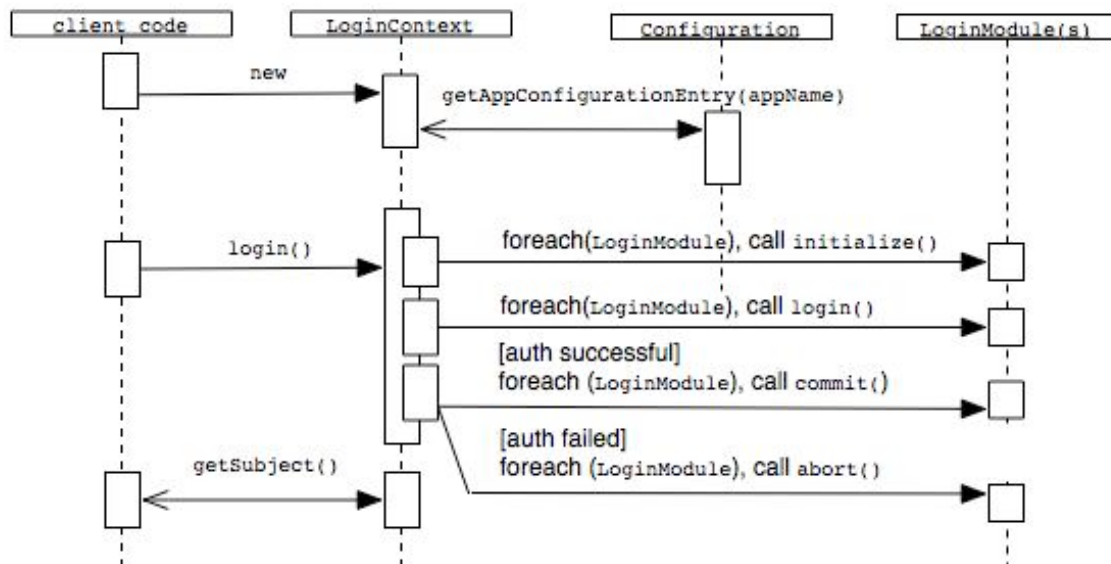
3 Authentication

The act of verifying the identity of a user, or “logging in,” is called authentication. When JAAS authenticates a `java.security.auth.Subject` it first verifies the user’s claims of identity by checking their credentials. If these credentials are successfully verified, the authentication framework associates the credentials, as needed, with the `Subject`, and then adds `Principals` to a `Subject`. The `Principals` represent any sort of identity the `Subject` has in the system, whether that identity is an “individual identity,” such as an employee number, or a “group identity,” such as belonging to a certain user group.

To perform the above functions, the `javax.security.auth.login.LoginContext` must be configured to use plug-in implementations of `javax.security.auth.spi.LoginModules`, usually provided by you. In addition to covering the above, this chapter will describe how to configure the `LoginContexts`, through both the standard flat-file, and also at runtime, programmatically.

3.1 Authentication Lifecycle

The diagram below illustrates the authentication lifecycle:



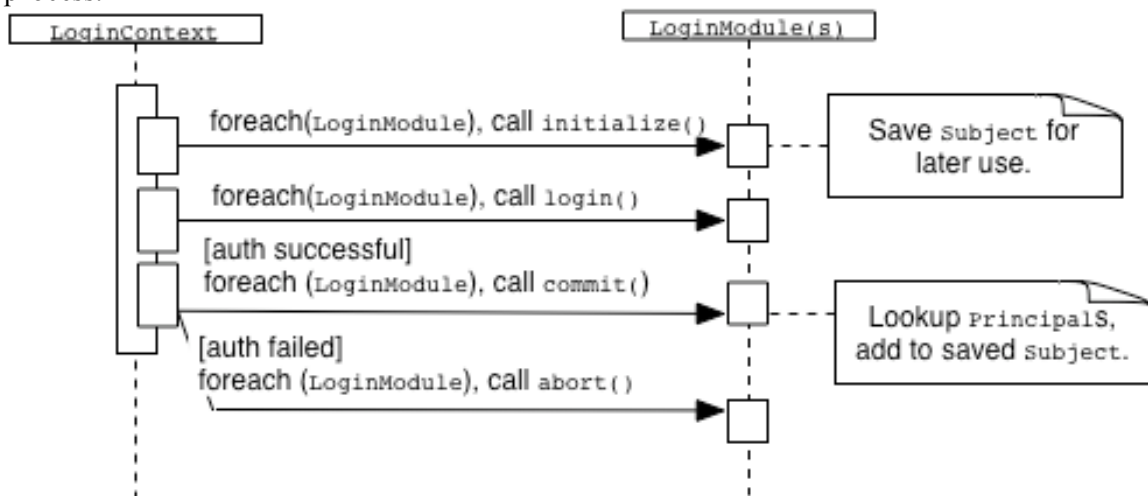
While the `javax.security.auth.login.LoginContext` interface contains only 3 methods, the lifecycle it goes through to authenticate a `Subject` is quite complex. A `LoginContext` is first created with at least two items: the “application name” it will be



authenticating Subjects for, and the `javax.security.auth.callback.CallbackHandler` to use for gathering credentials.

The “application name” is an arbitrary¹ name given to a set of one or more `LoginModules` that the `LoginContext` will delegate authentication to. This delegation allows JAAS to be both “pluggable” and “stackable.” It’s pluggable because anyone can write a `LoginModule` implementation and configure JAAS to use it, and stackable because multiple `LoginModules` can be used when authenticating a `Subject`. Configuring the `LoginContext` is done by static methods on the `javax.security.auth.login.Configuration` object, and by providing the singleton `Configuration` implementations that those methods use. Out of the box, the SDK uses a flat-file based `Configuration` implementation. A `Configuration`’s primary responsibility is answer the question, “for the specified ‘application name’ what `LoginModules` should a `LoginContext` use when authenticating a user?”

The `LoginContext` uses the list of `LoginModules` provided by the `Configuration`, following the `LoginModule` lifecycle diagramed above. More than one `LoginModule` may be used, allowing your application to have more than one source of authentication. For example, your application could consist of many sub-systems, each of which contributes their own `Principals` and, thus, different sets of permissions, to a `Subject`. Each `LoginModule` specified will first authenticate `Subject`, and, if the overall process was successful, add in any credentials and `Principals` needed to the `Subject`. The diagram below highlights this process:



Once all of the needed `LoginModules` have successfully verified a `Subject`’s claims of identity, the `LoginContext` has performed the bulk of its job. The same `LoginContext` used to authenticate a `Subject` is used to acquire the fully authenticated `Subject`. At the end of its lifecycle, the `LoginContext` can be used to log a user out of the system.

¹ The name is “arbitrary” because there is no special syntax for the name. By convention, new line characters and other white space characters are usually not used in an application name.



3.2 The LoginContext

A `javax.security.auth.login.LoginContext` instance is the controller used by JAAS to authenticate Subjects. LoginContexts are instance-based objects: a new instance is created each time you want to login or logout a Subject. Four constructors are provided:

```
LoginContext(String name)
LoginContext(String name, CallbackHandler callbackHandler)
LoginContext(String name, Subject subject)
LoginContext(String name, Subject subject, CallbackHandler
callbackHandler)
```

The second constructor is the one you'll typically use. The other constructors are there to provide you with the ability to control the Subject instance that will be used, or to use the default `CallbackHandler` instead of providing your own². Providing a Subject is useful for logging out users when you have a Subject but not the original `LoginContext` used to authenticate the Subject.

In addition to the constructors listed above, `LoginContext` has three methods:

```
getSubject()
login()
logout()
```

Before going in to details about those methods, we'll first take a look at the `CallbackHandler` interface and associated classes, as they are used through out the rest of the authentication lifecycle.

3.2.1 CallbackHandlers: Providing Credentials

The `CallbackHandler` is given the responsibility of gathering credentials for the Subject during authentication. A `CallbackHandler` is always associated with a `LoginContext` instance, and passed to the `LoginModules` that `LoginContext` controls. For example, a console-centric `CallbackHandler` may use “Username” and “Password” prompts to gather those two credentials; a Swing `CallbackHandler` may pop open a window to gather similar credentials; or, more common in web applications, the `CallbackHandler` will cache the username and password credentials entered with a request, pushing off the interactive part of gathering credentials to another part of the web application. In summary, all a `CallbackHandler` does is provide `LoginModules` with credentials when asked by the `LoginModules`.

The `CallbackHandler` interface contains just one method:

² The default `CallbackHandler` is specified by the security property `auth.login.defaultCallbackHandler`. This property is defined in the flat file `<JAVA_HOME>/lib/security/java.security`. In general, we recommend providing your own `CallbackHandler` instead of specifying it by security property.



```
handle(Callback[] callbacks)
```

The `Callback` interface itself has no methods. This seems peculiar at first, but after understanding of how credentials are designed in JAAS, a methodless `Callback` interface makes sense. Credentials themselves have no type in JAAS: they're simple `java.lang.Object` instance, or anything. The thinking behind this is that credentials can take any form, for example, from a simple `String` of a username and password, to a more complex object that represents a thumbprint.

As such, JAAS cannot place any limitations on what a `Callback` implementation must provide. Instead, the `CallbackHandler` implementation must know the type of the `Callback` and know how to handle instances of it. Similarly, when dealing with the credentials that a `CallbackHandler` provides, the `LoginModule` must know how to cast and deal with the credentials.

As the example in Chapter 1 showed, the `CallbackHandler` checks the type of each `Callback` passed into the `handle()` method. If the `CallbackHandler` recognized the type, casts it and fills in the `Callback` details as needed:

```
public void handle(Callback[] callbacks) {
    for (int i = 0; i < callbacks.length; i++) {
        Callback callback = callbacks[i];
        if (callback instanceof NameCallback) {
            NameCallback nameCB = (NameCallback) callback;
            nameCB.setName(username);
        } else if (callback instanceof PasswordCallback) {
            PasswordCallback passwordCB = (PasswordCallback) callback;
            passwordCB.setPassword(password.toCharArray());
        }
    }
}
```

While `handle()` can throw an `UnsupportedCallbackException` if the `Callback` passed in not supported by the `CallbackHandler`, we favor simply not filling out the `Callback`, and allowing the `LoginModule` to fail to login the user. If you choose to follow this convention, instead of throwing an `UnsupportedCallbackException`, a warning message should be logged.

As you can imagine, there can be a large degree of difference between various `CallbackHandler` and `Callback` implementations. Later in this book, in chapter XXX, we'll go over several best practices and idioms for implementing the two interfaces.

Callbacks for Name and Password

J2SE ships with several `Callbacks`, two of which will be particularly useful to you:

```
javax.security.auth.callback.NameCallback
javax.security.auth.callback.PasswordCallback
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

As the class names suggest, the `NameCallback` provides a JavaBean-style property for the name, or username, of the `Subject` authenticating, while the `PasswordCallback` provides a property for a password. In most cases, these two `Callbacks`, and the credentials they provide, will suffice for authenticating a user. As such, most of the `CallbackHandlers` you write will probably support the `NameCallback` and `PasswordCallback`.

`NameCallback`

In addition to the `getName()` and `setName()` property methods, `NameCallback` provides a method, and constructor argument for setting the `String` “prompt.” In those cases where your `CallbackHandler` will interact with the user to gather credentials, this prompt is used to ask the user to enter their name. When the `CallbackHandler` is not performing this user-interaction, the prompt can simply be set to null, or any value, and ignored by the `CallbackHandler`.

`PasswordCallback`

The `PasswordCallback` contains several methods to help deal with the sensitive nature of passwords. For example, the `clearPassword()` method erases the value of the password property. Once a `Subject` has been authenticated, this method should be called to ensure that a malicious piece of code can’t call `getPassword()` on the `PasswordCallback` handler. That is, this method is used to minimize the time the clear-text password is available in the VM.

3.3 *LoginModules*

Implementations of `LoginModules` provide the core of JAAS authentication. Though `LoginContext` provides the client interface for JAAS authentication, `LoginContext` acts as a controller, delegating the majority of the authentication work and decisions to the list of `LoginModules` configured.

Three design goals drive the interface and implementation contract for `LoginModules`:

1. `LoginModules` should be “plugin-able.”
2. `LoginModules` should be “stackable,”
3. As a consequence of being stackable, `LoginModules` should follow a two-phase commit cycle.

Being plugin-able means that the same `LoginModule` implementation can potentially be re-used in different applications, and added to an application without having to recompile code. For example, a `LoginModule` that authenticates users for in Windows Domains or Active Directory, could be provided by a third party for use in other applications.

`LoginModules` are “stackable” because multiple `LoginModules` can be used to authenticate one `Subject`. You may need to use more than one `LoginModule` because your application may have more than identity management system. For example, your application could be an employee records system that needs to authenticate with the HR system to access insurance records and the payroll system to access salary records. Each of these two systems could require a user to login, contributing `Principals` to the `Subject`, and thus granted the required `Permissions`.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

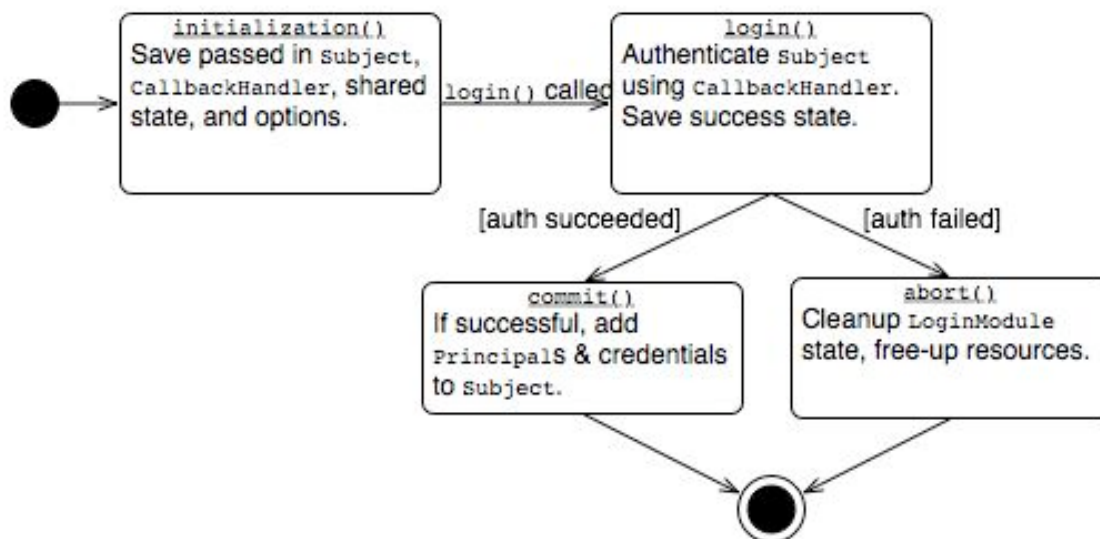
License: <http://creativecommons.org/licenses/by-nc/2.5/>

When more than one `LoginModule` is used, a certain degree of transaction management is implicitly required. If your application is using two `LoginModules` to authenticate Subjects, if an error occurs in either, or if either fails to authenticate a user, the positive effects of the other should not occur to the Subject. For example, if the first `LoginModule` successfully logs a user in, but the second does not, the Subject being authenticated should not get the Principals that the first `LoginModule` would assign the Subject. In fact, `LoginModules` can be configured to be either strictly required, or completely optional.

The job of a `LoginModule` is simple: use credentials to verify a Subject's identity and then associate the appropriate Principals and credentials with that Subject. To enable the transactional benefits of a two phase commit process, however, things are complicated by the need for a lifecycle.

3.2.2 *LoginModule Life-Cycle*

Below is an activity diagram of a `javax.security.auth.spi.LoginModule` implementations' lifecycle:



First, the `LoginModule` implementation's default constructor is used to create a new instance of the `LoginModule`. Because the default, no argument constructor is used, another method is used to pass in the objects the `LoginModule` will use. This is done with the `initialize()` method, which takes the Subject to be authenticated, the `CallbackHandler` to use to gather credentials, a `Map` used as a session shared by all the `LoginModules` in use, and a `Map` of configuration options specified by the `Configuration`.

When the `LoginContext` executes a `LoginModule`'s `login()` method, the `LoginModule` does whatever is needed to authenticate the Subject. This is the first phase of the two-phase process. If the login attempt succeeds, `true` is returned; if it failed, a `javax.security.auth.LoginException`, or one of its sub-classes, is thrown; if the `LoginModule` should be ignored [for what reason?], `false` is returned. Each `LoginModule` stores whether it succeeded or not as private state, accessible by the other methods during the authorization lifecycle. Notice that Principals and Credentials are not yet added to the Subject in the `login()` method.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

Once all the `LoginModules` required to be successful have succeeded, the `LoginContext` controller calls the `commit()` method on each `LoginModule`. In the `commit()` method, the `LoginModule` will access the private success state. If authentication succeeded, the `commit()` method adds `Principals` and `Credentials` to the `Subject` and does any cleanup needed; if unsuccessful, just clean-up is done.

If login wasn't successful, the `LoginModule`'s `abort()` method is called instead of `commit()`. Execution of the `abort()` method signals that the `LoginModules` should cleanup any state kept, and assure that no `Principals` or `Credentials` are added to the `Subject`.

Next, we go over each of these steps, and implementing them, in detail.

3.2.3 Configuration, Creation, and Initialization

As summarized above, the `LoginModules` are configured in groups by the `javax.security.auth.login.Configuration` service. The `LoginContext` authenticates `Subjects` using these groups. By default, a flat-file based `Configuration` is used, which is sufficient for our examples. Chapter 4 introduces a more dynamic, runtime `Configuration`. A `Configuration` defines several “stacks” of `LoginModules`, each given a name. The term stack is often used because the order each `LoginModule` is specified in is important: they'll be used in that order. JAAS refers to this an “application.” Another way to think of these stacks and application names is as “`LoginModule` groups” and “group names.”

The below login module file, in the syntax of the default `Configuration` implementation, shows several examples of these `LoginModule` groups:

```
simple
{
    auth.SimpleLoginModule REQUIRED;
};

multipleSources
{
    auth.WindowsDomainLoginModule REQUIRED;
    auth.SolarisLoginModule OPTIONAL;
    auth.CustomSystemLoginModule OPTIONAL;
}
```

In the above example, the “simple” `LoginModule` application, or group, consists of just one `LoginModule` that must pass for a `Subject` to be fully authenticated. A single `LoginModule` group like this is what you'll typically use for self-contained applications. The second group contains three `LoginModules`, only one of which is required. A group like this is more typical in an application that integrates with several other systems and applications. [More explanation of an example where this might occur?]

Control Flags

Each stack of `LoginModules` in a `Configuration` is given a “success acceptability” control flag. This flag determines how the success or failure of a `LoginModule` effects the over-all success of the authentication attempt. The 4 possible states of “success acceptability” are:



- **required** – the `LoginModule` must succeed. That is, it must return `true` from the `login()` method. However, regardless of success, the `LoginContext` continues calling the `login()` method on the rest of the `LoginModules`.
- **requisite** – the `LoginModule` must succeed: it must return `true` from the `login()` method. Unlike **required** `LoginModules`, the failure of a **requisite** `LoginModule` prevents the `login()` method of the remaining `LoginModules` from being called.
- **sufficient** – the `LoginModule` isn't required to succeed. But, if it does succeed, no other `LoginModules` are called. That is, once a **sufficient** `LoginModule` returns `true` from its `login()` method, no other `login()` methods will be called.
- **optional** – success isn't required for **optional** `LoginModules`. Whether an **optional** `LoginModule` is successful or fails, the authentication still goes down the stack.

The overall success of a group of `LoginModules` is determined by the collective success as outlined in the above. If successful, the `commit()` method will be called on all `LoginModules`. Otherwise, the `abort()` method is called, signaling to all `LoginModules` that the overall authentication process failed.

Creation and Initialization

Typically, and definitely with the default Configuration, `LoginModules` are created using the default, no argument constructor. As such, instead of performing instance initialization in the constructor, the `initialize()` method should be used. The `initialize` method will be called before the `LoginModule`'s other methods are called. Four parameters are passed into the method:

```
public void initialize(Subject subject, CallbackHandler handler, Map
sharedState, Map options)
```

A `LoginModule` implantation should store each of these items as private session state, as the other methods will need to access them. Each object passed in is shared between all `LoginModules` in a `LoginModule` group, so care must be taken not to ruin them for the others, for example, removing all the entries from the `sharedState` Map.

LoginModule Options

In addition to configuring the control flag, an optional Map of configuration “options” is passed into the `initialize` method. The contents of this Map aren't defined, but with the default Configuration, the Map contains entries of `String` keys and `String` values. With the default Configuration implementation, these options are configured in as name/value pairs after the control flag. Building on the above flat-file example:

```
multipleSources
{
    auth.WindowsDomainLoginModule REQUIRED debug="true";
    auth.SolarisLoginModule OPTIONAL;
    auth.CustomSystemLoginModule OPTIONAL setCookie="false",
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5
License: <http://creativecommons.org/licenses/by-nc/2.5/>


```
caching="session";  
}
```

When the initialize method for the above CustomSystemLoginModule is called, you'd access these options like this:

```
// from chp03.ExampleLoginModule  
public void initialize(Subject subject,  
    CallbackHandler callbackHandler, Map sharedState, Map options) {  
    // store other args as member fields  
    this.debug = Boolean.valueOf((String)options.get("debug"));  
    this.caching = options.get("caching");  
    // do other setup  
}
```

If you provide an alternative to the file-based Configuration, it's a good idea to keep the contents of the options Map as String name/value pairs. This assures that your LoginModule is easily plug-able into a wider range of Configuration implementations, such as the default Configuration.

Shared State

To facilitate coordination among the LoginModule instances in a group of LoginModules, a Map known as the shared state is passed to each LoginModule. Because of the sequential nature of executing the LoginModule stack, the Map is effectively thread-safe³. But, since the same Map is passed to each, each LoginModule could ruin the Map for the others.

The exact use of the sharedState isn't specified, much like the exact way to use an HttpSession is left up to the end-users. One use, for example, might be to store credentials like username and password that other LoginModules have gathered. Instead of having to request them from a user again, other LoginModules can first attempt to pull them from the shared state.

login()

Once the LoginModule has been initialized, the login() method is called. In this method, the LoginModule authenticates the user, but doesn't yet modify the Subject. The login() method implementation typically creates Callbacks, and passes them to the CallbackHandler's handle() method. Once the credentials are gathered, the login() method is responsible for verifying the credentials, for example, by comparing a username and password to those stored in a database.

The login() method below shows a simple, but typical implementation:

```
// from chp03.ExampleLoginModule  
public boolean login() throws LoginException {  
    NameCallback name = new NameCallback("Username:");
```

³ Of course, if a LoginModule's code passes the Map to another thread that's running concurrently to the LoginContext, the use of the Map could become un-thread-safe.



```
        PasswordCallback password = new PasswordCallback("Password",
            false);
    try {
        handler.handle(new Callback[] { name, password });
    } catch (IOException e) {
        LoginException ex = new LoginException(
            "IO error getting credentials: " + e.getMessage());
        e.initCause(e);
        throw ex;
    } catch (UnsupportedCallbackException e) {
        LoginException ex = new LoginException(
            "UnsupportedCallback: " + e.getMessage());
        e.initCause(e);
        throw ex;
    }
}

String usernameText = name.getName();
String passwordText = String.valueOf(password.getPassword());

userAuthenticated = UserService.checkPassword(usernameText,
    passwordText);

if (userAuthenticated) {
    username = usernameText;
    return true;
} else {
    throw new FailedLoginException("Username/Password for "
        + usernameText + " incorrect.");
}
}
```

Let's take a look at each step in this implementation.

Gathering Credentials with Callbacks

This `login()` implementation is only concerned with two credentials, username and password. The first thing it does is create `NameCallback` and `PasswordCallback` instances:

```
NameCallback name = new NameCallback("Username:");
PasswordCallback password = new PasswordCallback("Password:",
false);
```

The first argument passed to each is the prompt to use when interactively gathering the credentials. The `PasswordCallback` takes a second argument, whether to echo the password entered or not. Typically, you'll want this to be `false`, as displaying a password is a bad idea. As mentioned above, interactive gathering is typically done for single-user, desktop



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5
License: <http://creativecommons.org/licenses/by-nc/2.5/>

applications. When a `LoginModule` is used in web applications, `Callbacks` and `CallbackHandlers` will typically not be given the responsibility to gather credentials from the user. A form on a JSP page will collect the credentials, and your `CallbackHandler` will be used more like a data transport object between the JSP page and JAAS's authorization framework. We'll see an example of this type of `CallbackHandler` in the next chapter.

Once the `Callbacks` have been created, they're passed to the `CallbackHandler` that attempts to fetch the credentials represented by the `Callback`.

Using the CallbackHandler

```
// from chp03.ExampleLoginModule#login()
try {
    handler.handle(new Callback[] { name, password });
} catch (IOException e) {
    LoginException ex = new LoginException(
        "IO error getting credentials: " + e.getMessage());
    e.initCause(e);
    throw ex;
} catch (UnsupportedCallbackException e) {
    LoginException ex = new LoginException(
        "UnsupportedCallback: " + e.getMessage());
    e.initCause(e);
    throw ex;
}
```

Though the above code revolves around just one line, the bulk of the code is error handling. First, we create an anonymous, inline array of the `Callbacks` that specify the credentials we need. `CallbackHandler`'s interface doesn't guarantee that the `Callbacks` will be used in the order that they appear in the array, but the convention is to simply iterate through them in the array order. As such, you would typically put a username `Callback` before a password `Callback`. However, there is no guaranteed that the `CallbackHandler` will respect the order of the array.

Indeed, a `CallbackHandler` may not support a `Callback` passed to its `handle()` method. In such cases, the `CallbackHandler` may either throw an `javax.security.auth.callback.UnsupportedCallbackException`, or simply ignore the `Callback`. Our recommendation is to simply ignore the `Callback`, at most logging a warning message. This strategy allows for a more tolerant `CallbackHandler` that can more easily be re-used and plugged into to different `LoginModule` groups.

A `java.io.IOException` may also be thrown from the `CallbackHandler` if an error occurs acquiring the credentials. As with other design features of JAAS authentication, throwing an `IOException` really makes sense only for desktop application, where the `CallbackHandler` will actually be responsible for prompting the user for credentials. In such cases, the `CallbackHandler` may write out the prompt to `System.out`, for example, and then encounter an `IOException` reading from `System.in`.

In both cases, in our example code if an `UnsupportedCallbackException` or `IOException` is thrown, a `LoginException` wraps the exception. Because the



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

LoginException constructors don't take a cause exception (functionality which wasn't introduced until J2SE 1.4), we call the `initCause()` method to record the cause.

Verifying Credentials

```
// from chp03.ExampleLoginModule#login()
String usernameText = name.getName();
String passwordText = String.valueOf(password.getPassword());

userAuthenticated = UserService.checkPassword(usernameText,
    passwordText);

if (userAuthenticated) {
    username = usernameText;
    return true;
} else {
    throw new FailedLoginException("Username/Password for "
        + usernameText + " incorrect.");
}
```

Once the `CallbackHandler` has filled out the `Callbacks`, we're ready to verify those credentials with our authentication source. In our example, we have a service method `UserService.checkPassword()` that simply returns true if the passed in credentials match what's stored in the database. The success of this verification is saved in the private member Boolean field, `userAuthenticated_`. This field is used by other methods to signal if the user was successfully authenticated.

If the user did successfully authenticate, the `login()` method first saves the username entered as a private field for later use to create `Principals` and credentials, and then returns true. Otherwise, a sub-class of `LoginException`, `javax.security.auth.login.FailedLoginException`, is thrown. Returning false from the `login()` method signals to JAAS to ignore this `LoginModule`. You may want the `LoginModule` ignored if it has nothing to contribute to the Subject. Typically, these `LoginModules` will be configured with the control flag `optional`.

3.2.4 commit()

If the Subject has authenticated as required by the `LoginModule` group's collective control flags, the `commit()` method will be called on each `LoginModule`. The `abort()` method is called if authentication didn't succeed. For example, if there are 3 `LoginModules` in a `LoginModule` group, one required, and two optional, as long as the required `LoginModule` returns true from `login()`, the authentication process will call `commit()` on each `LoginModule`, regardless of the optional `LoginModules` success.

Our example `commit()` method is below:

```
// from chp03.ExampleLoginModule
public boolean commit() {
    if (userAuthenticated) {
        Set groups = UserService.findGroups(username);
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
for (Iterator itr = groups.iterator(); itr.hasNext();) {
    String groupName = (String) itr.next();
    UserGroupPrincipal group = new UserGroupPrincipal(groupName);
    subject.getPrincipals().add(group);
}

UsernameCredential cred = new UsernameCredential(username);
subject.getPublicCredentials().add(cred);
}
// either way, cleanup
username = null;
return true;
}
```

First, `commit()` checks the private field `userAuthenticated` to see if authentication was successful in the `login()` method. If it was, the `commit()` method looks up the groups that the Subject is a member of, and then creates a `UserGroupPrincipal` for each, adding them to the Subject. Next, the `commit()` method adds a credentials for the username. The class `UsernameCredential` is another custom class that simply wraps the username, providing a `getUsername()` method as well as `equals()` and `hashCode()` implementations. You'll typically want to store at least the username credential for looking up who the user is later in the application.

Finally, whether or not authentication was successful, the username credential is cleared out. You should clear out any other credentials you've stored at this point as well. Additionally, to keep sensitive information from lingering in memory, where it could be compromised by malicious code while waiting to be garbage collected, you should null-out references to other fields.

Once the `commit()` method is done, it returns `true` to indicate that everything went well. If an error occurs in `commit()`, a `LoginException` may be thrown. If this `LoginModule` should be ignored, as with the `login()` method, `commit()` should return `false`.

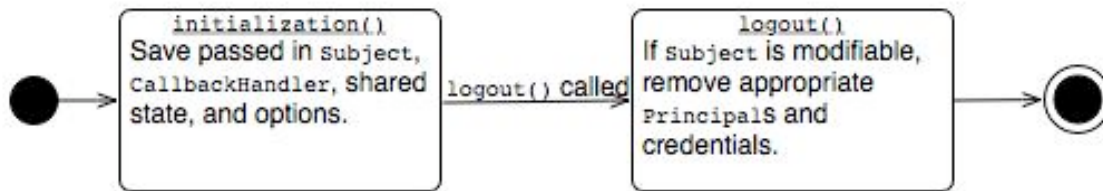
3.2.5 *abort()*

```
// from chp03.ExampleLoginModule
public boolean abort()
{
    username = null;
    subject = null;
    return true;
}
```

The `abort` method is called when overall authentication fails. In the `abort()` method you should cleanup any member fields to remove state, and prevent malicious code from accessing potentially sensitive information. If an error occurs, `abort()` can throw a `LoginException`.



3.2.6 *logout()*



The `logout()` method is called when the `LoginContext`'s `logout()` method is called. This is usually done much after the `login()/commit()/abort()` cycle. A `LoginModule`'s `logout()` method should remove any Principals and credentials it added to the Subject. For example:

```
// from chp03.ExampleLoginModule
public boolean logout() {
    if (!subject.isReadOnly()) {
        Set principals = subject
            .getPrincipals(UserGroupPrincipal.class);
        subject.getPrincipals().removeAll(principals);
        Set creds = subject
            .getPublicCredentials(UsernameCredential.class);
        subject.getPublicCredentials().removeAll(creds);
        return true;
    } else {
        return false;
    }
}
```

In the above example, we first check that the Subject is modifiable by calling `isReadOnly()` on Subject, avoiding an exception being thrown when we attempt to modify a read only Subject. If the Subject is modifiable, we use the principal and credential query methods on Subject to retrieve the items the `ExampleLoginModule` added (in the `commit()` method) to the Subject. In general, you shouldn't rely on the `LoginModule`'s instance state – fields you set in the `login()` or `commit()` methods – because you're not guaranteed to have the same `LoginModule` instance when `logout()` is called as when the other methods were called. For example, a new `LoginModule` instance could be created when when it's time to log out a Subject.

Because of this inability to guarantee that the `LoginModule` is the same instance, you may find it difficult to figure out which Principals and credentials to remove from the Subject in the `logout()` method. One design tactic to get around this is to associate Principal and credentials implementation with specific `LoginModule` implementations. That is, in your system, a specific `LoginModule` can only ever add a specific types of Principal and credentials to a Subject. This way, a `LoginModule` can always remove Principals as is done above, by getting the Set of Principals by type, and then calling `removeAll()` on the Subject's Principal Set.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

3.3 Subject

Once the `LoginContext`, and the group of `LoginModules` it delegates to, have authenticated a `Subject`, you can retrieve the `Subject` by calling the `getSubject()` method of `LoginContext`. The `Subject` class contains two types of methods: static, `doAs` methods to help execute code with a `Subject`'s privileges, and instance methods to retrieve and modify the `Subject`'s state. We cover the `doAs` methods in more detail in chapter XXX, so here we just go over the second type of methods.

The `Subject` aggregates three things: `Principals`, public credentials, and private credentials. Each of these aggregated items can be retrieved through one of two methods: one version returns all the items, while the other methods filters based on the `java.lang.Class` type of the item. We saw the second type of this method in use in our `ExampleLoginModule`'s `logout()` method.

3.3.4 Principals

```
getPrincipals()  
getPrincipals(Class)
```

The two `Principal` methods both return `Sets` of `Principals`. The no argument method returns all of the `Subject`'s `Principals`, while the second returns only those `Principals` that are instances of subclasses of the passed in `java.lang.Class`.

3.3.5 Credentials

```
getPublicCredentials()  
getPublicCredentials(Class)  
getPrivateCredentials()  
getPrivateCredentials(Class)
```

A `Subject`'s credentials are divided into two types: public and private. As their names imply, the simple rule of thumb is that credentials that could safely accessible to anyone are public, while all other credentials are private. Usernames are typically public, while passwords are certainly private.

As with the `Principal` methods, credentials can either be retrieved all at once with the no argument methods, or filtered using a passed in `java.lang.Class`. When a `Class` is passed in, the credentials in the returned `Set` will be either instances or subclasses of the passed in `Class`.

3.3.6 Read Only State

Finally, the `Subject` provides two methods to set and query for the `Subject`'s modifiability:

```
isReadOnly()  
setReadOnly()
```



The `isReadOnly()` method is a typically JavaBean read get-method, returning `true` if the `Subject` is read only, and `false` otherwise. Once a `Subject`'s `setReadOnly()` method has been called, the `Subject` cannot be made writeable again.

Summary

This chapter introduced the JAAS classes used to authenticate, or "log in" users. In the opening sequence diagram, and following discussion, we saw that the `LoginContext` is used as a controller to coordinate the use of the other classes such as `Configuration`, `LoginModules`, and as the glue for putting together the other authentication classes. We also covered `LoginModule`'s life-cycle in-depth and provided a simple example of implementing a `LoginModule`. Finally, we discussed some of the finer methods available on `Subject`, along with how and why you might use them.

