Welcome!

If you're interested in this book, you have some very obscure interests. Nonetheless, here it is, my overview of the Java Authentication and Authorization Service.

This book is from 2005, so it's very out of date.

I originally worked on this for Manning, but reviewers said they wouldn't pay $50 for it, so it was canceled. So, here it is!

The source code is here: https://github.com/cote/jaasbook

I don't remember anything about JAAS, so sadly I can't answer any questions. You're on your own!

@cote

Amsterdam, April 2019

# 1. Introducing JAAS

JAAS, the Java Authentication and Authorization Service, has been a standard part of the Java security framework since version 1.4 version and was available as an optional package in J2SE 1.3. Before that time, the previous security framework provided a way to allow or disallow access to resources based on what code was executing. For example, a class loaded from another location on the Internet would have been considered less trustworthy and disallowed access to the local file system. By changing the security settings, this restriction could be relaxed and a downloaded applet could modify local files. Viewed mainly as a tool for writing clients, early Java didn't seem to need much more than this.

As the language matured and became popular for server-side applications as well, the existing security framework was deemed too inflexible, so an additional restriction criterion was added: *who was running the code*. If User A was logged in to the application, file access may be allowed, but not so for User B. Accomplishing this requires authentication and authorization. *Authentication* is the process of verifying the identity of users. *Authorization* is the process of enforcing access restrictions upon those authenticated users. JAAS is the subsection of the Java security framework that defines how this takes place.

Like most Java APIs, JAAS is exceptionally extensible. Most of the sub-systems in the framework allow substitution of default implementations so that almost any situation can be handled. For example, an application that once stored user ids and passwords in the database can be changed to use Windows OS credentials. Java's default, file-based mechanism for configuration of access rights can be swapped out with a system that uses a database to store that information. The incredible flexibility of JAAS and the rest of the security architecture, however, produces some complexity. The fact that almost any piece of the entire infrastructure can be overridden or replaced has major implications for coding and configuration. For example, every application server's JAAS customizations have a different file format for configuring JAAS, all of which are different from the default one provided by Java[1].

## 1.1 User Access Control

Suppose you're tasked with writing a web application that allows users to log in with an id and password and then allows the users to view their employment information. Because of the sensitivity of the data, it is important that employees not have access to each other's data. At this point, the protection logic is not very complex: only let the user that is currently logged in see the information that is mapped to himself. But now add the idea of a manager who may be able to see some of the other employees' items, such as salary or hiring date. Then add the idea of a human resources administrator. Then an accountant. Or an auditor.

1 But, we're Java programmers, we live for that kind of stuff, right?

The CEO. All these users need access to different information and should not be allowed anything more than necessary.

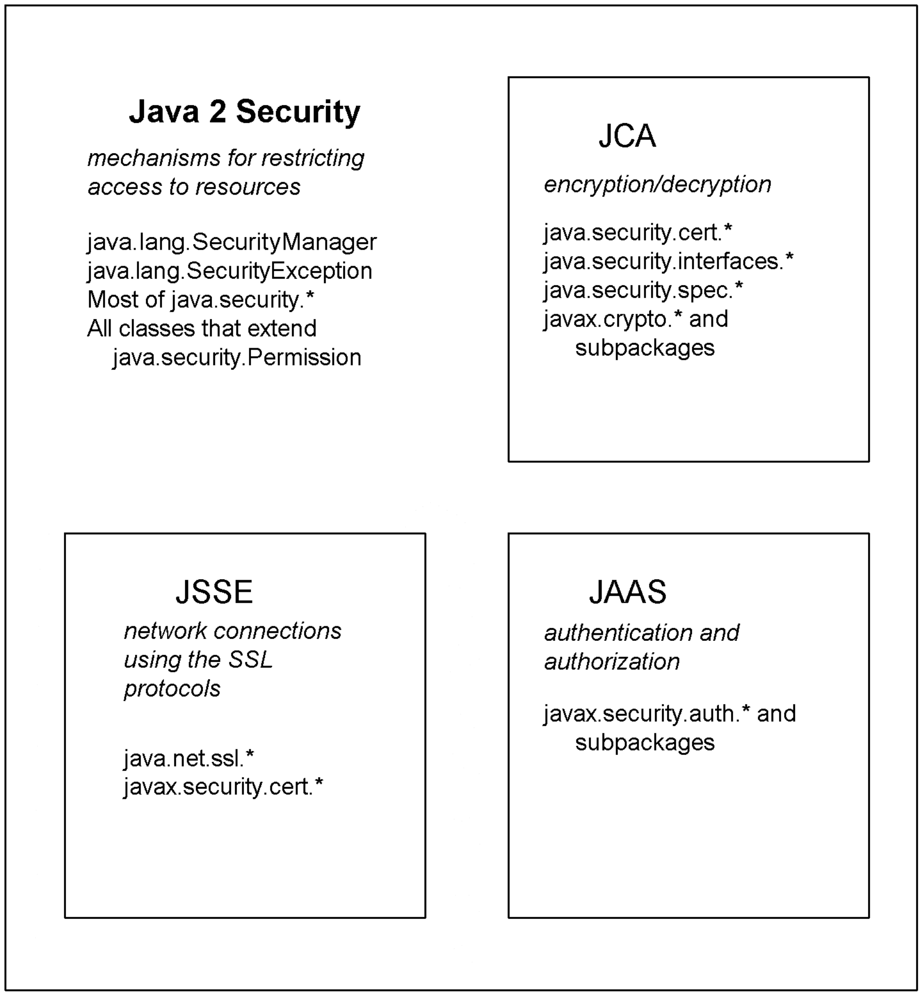
Complex security domains like these are where JAAS comes in handy. Additionally, most application servers and servlet containers use JAAS to provide a way to pass login information into the application. The application can even take advantage of login methods provided by the application server and never deal with the interaction with the user.

By the end of this book, you will both understand and use the functionality in JAAS, and also be able to replace many of the pieces provided by the JDK or whatever application server you may be using with your own custom classes. The rest of this chapter covers high levels security concepts, narrowing down to JAAS at the end.

## *1.2 The Java 2 Security Architecture*

The main functionality of the Java 2 security architecture is protecting resources. "Resources" can be anything, but are usually some chunk of data: employee records, databases, or more abstract pieces of data such as class files. The classes in the `java.security` package do that work directly by defining the process for testing access permission, and associating permissions with code based on the source from which it was loaded. There are additional subsections and utilities also included in the architecture:

- The Java Cryptography Architecture (JCA) defines interfaces and classes for encrypting and decrypting information.
- The Java Secure Socket Extension (JSSE) uses the JCA classes to make secure network connections.
- JAAS, the topic of this book, which performs authentication and authorization.



Java 2 Security
mechanisms for restricting access to resources
java.lang.SecurityManager
java.lang.SecurityException
Most of java.security.*
All classes that extend java.security.Permission
JCA
encryption/decryption
java.security.cert.*
java.security.interfaces.*
java.security.spec.*
javax.crypto.* and subpackages
JSSE
network connections using the SSL protocols
java.net.ssl.*
javax.security.cert.*
JAAS
authentication and authorization
javax.security.auth.* and subpackages

## *1.3 JAAS*

This book is concerned primarily with the lower right box in the diagram above: JAAS. JAAS is a mix of classes specific to JAAS, and classes "borrowed" from other parts of the Java security framework. The primary goal of JAAS is to manage the granting of permissions and performing security checks for those permissions. As such, JAAS is not concerned with other aspects of the Java security framework, such as encryption, digital signatures, or secure connections.

There are several concepts and components that make up JAAS, but all of them revolve around one part: permissions.

### *1.3.1 Permissions*

A *permission* is a named right to perform some action on a target. For example, one permission might be "all classes in the package `com.myapp` can open sockets to the Internet address `www.myapp.com`."

## *Who is Granted a Permission?*

As the example implies, some "entity" is granted a permission. In Java, this entity is usually either a user or a "code base." Users are a familiar concept, and generally map to a person or process executing code in your application. Users as entities are discussed at length below in the sections on `Subjects` and `Principals`. On the other hand, a code base is a bit more vexing in it's meaning. A code base is a group of code, usually delineated by a JAR or the URL from which the code was physically loaded. For example, all the classes downloaded as an applet from a remote server could be put into a single code base. Then, because permissions can be applied to code bases, your application could disallow code from that applet from accessing the local file system. You would want to deny access to the local file system to, for example, prevent applets from installing spy- or ad-ware on your machine, or installing viruses. This ability to "sandbox" code, keeping remote, un-trusted code from performing malicious actions, was one of the prime selling points of Java early on.

In Java, sub-classes of the abstract `java.security.Permission` class are used to represent all permissions. There are several types of permissions shipped in the SDK, such as `java.io.FilePermission` for file access, or `java.net.SocketPermission` for network access. The special permission `java.security.AllPermission` serves as a stand-in for any permission. Additionally, you can create any number of custom permissions by extending the class yourself.

A `Permission` is composed of three things, only the first two of which are required:

1. The type of `Permission`, implicit in its class-type.
2. The `Permission`'s name, generally the target(s) the `Permission` controls.
3. Actions that may be performed on the target.

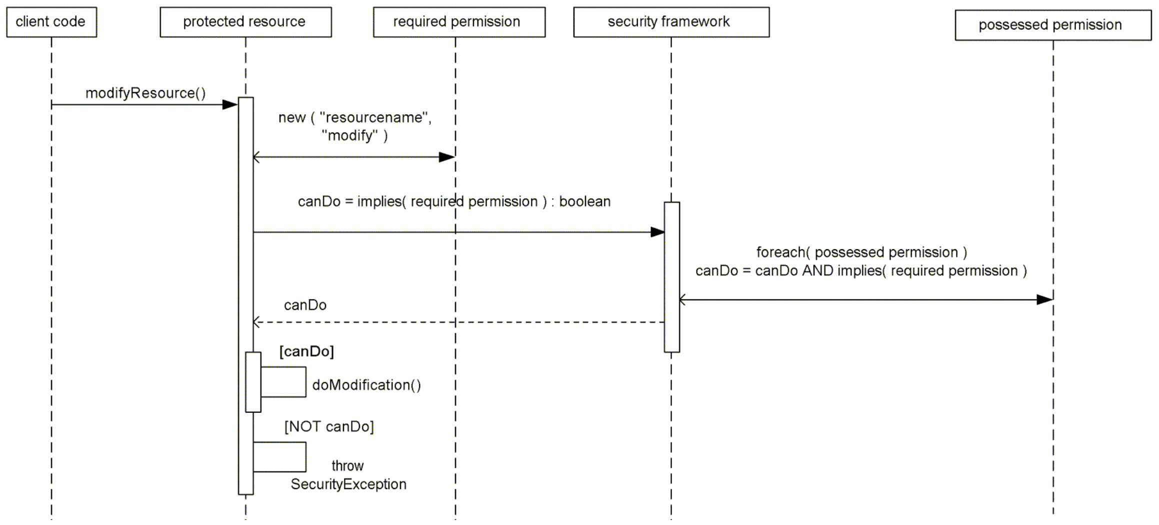
Conceptually, the type and the name of the `Permission` specify what is being accessed. The actions are generally a comma-delimited set of allowed actions. A `FilePermission` named "`pristinebadger.doc`" with actions "`read, write`" would let the possessor read from and write to the file "`pristinebadger.doc`". The table below illustrates a 4 more examples:

| **Permission Type** | **Target** | **Action** |
|---|---|---|
| `ProfilePermission` | All | read |
| `ProfilePermission` | All | write |
| `DocumentPermission` | "Project Avocado" | read |
| `DocumentPermission` | All "programming group" documents | write |

The actual "allowing" takes place in the `boolean implies( Permission )` method in `Permission`. When resource access occurs, the resource constructs an instance of the corresponding `Permission` class and passed it to the security framework. The framework then tests if the current security context[2] has been granted the right(s) described by the `Permission` instance. The security framework searches for a permission with the correct type and name. If it finds one, it calls `implies` on it, passing in the newly constructed

2 The "security context" includes the rights granted to the currently executing code and, if available, the currently logged in user.

`Permission` instance. If `true` is returned, access is allowed. The fact that the instance of `Permission` performs the check means that custom permissions can be created. The diagram below illustrates this process:



**The general flow of permission checking**

## *Permissions are positive*

`Permissions` express what *can* be done, not what *cannot* be done. First, this design decision helps avoids any conflict in permissions. For example, if a user has a permission that gives it access to the C: drive, and another permission that expressly forbids access the C: drive, which one applies? Some method of resolving problems such as this would need to be part of the security model. Instead of devising a conflict resolution process, Java defines only positive permissions. When permissions can only express the positive any chance of conflict is removed.

Alternatively, the creators of JAAS could have chosen to express only negative permissions. A permission would be granted unless it was expressly forbidden. This opens up a security hole in that you must know about everything you want to restrict ahead of time. For example, if a new file enters the system, users will by default have access to that file until the system expressly forbids access. In the mean time, a malicious user could access the file. With a positive permission model, JAAS avoids this need to express everything that is forbidden and the problems that can arise in such a system.

## *1.3.2 The SecurityManager and AccessController*

The pre-JAAS security model employed the concept of a security manager, a singleton of `java.lang.SecurityManager` through which all the types of permission were expressed and checked. This class is still used in current versions of Java as the preferred entry point for security checks, but it has traces of the pre-JAAS security model. A quick inspection of the `java.lang.SecurityManager` reveals methods with names like `checkDelete` and `checkExec`. These correspond to each task that needs explicit permission to be performed, such as deleting a file or creating an exec process. Each of these methods will throw a `SecurityException` if the permission in question has not been granted. If the permission has been granted, the method simply returns. The code guarding the resource would call the specific `check` method either granting access, or throwing a `SecurityException` if permission is not granted[3].

For example, the code in `java.io.File` delete may have included something like:

```
public void delete( String filename ) {
  …
 SecurityManager.getInstance().checkDelete(filename)
  // no SecurityException thrown, so delete file
  …
}
```

If permission has been granted to delete `filename`, the call to `checkDelete` will pass, and the file will be deleted by additional code. If permission has not been granted, `checkDelete` will throw a `SecurityException`, meaning that the code calling delete will need to catch that exception and respond accordingly (perhaps by displaying an error message to the user).

A developer wanting to protect a new resource would have to extend the `SecurityManager`, create new `checkXXX` methods, and tell the VM to use this new `SecurityManager` using the runtime property, `java.security.manager`. This unwieldy option spurred the creation of the `Permission` class in Java 2. Also, a new method, `checkPermission`, could now be used to check any `Permission`, enabling the creation of new permissions without having to create a new `SecurityManager`.

To back this new model, a new permission checking service class was introduced `java.security.AccessController`. For backwards compatibility, the existing `SecurityManager` was preserved and, as mentioned above, is still the preferred entry point for permission checking. Like `SecurityManager`, `AccessController` also has a `checkPermission` method. In fact, the default implementation of `SecurityManager` delegates its calls to `AccessController`. This class knows how to access the current thread's security-related context information, which includes all the permissions allowed for the code-stack that is currently executing. A call to `AccessController.checkPermission()` thus extracts those permission and checks the supplied permission against each piece of code executing on the stack.

## *Enabling the SecurityManager*

By default, the `SecurtyManager`, and thus security as a whole, is disable when you run Java. The `SecurityManager` can be enabled with the VM argument
`-Djava.security.manager`, or by setting the `SecurityManager` to use programmatically

[3] This model of security checking is why so many methods in the JDK throw `SecurityException`.

at runtime.

## 1.3.3. Policies

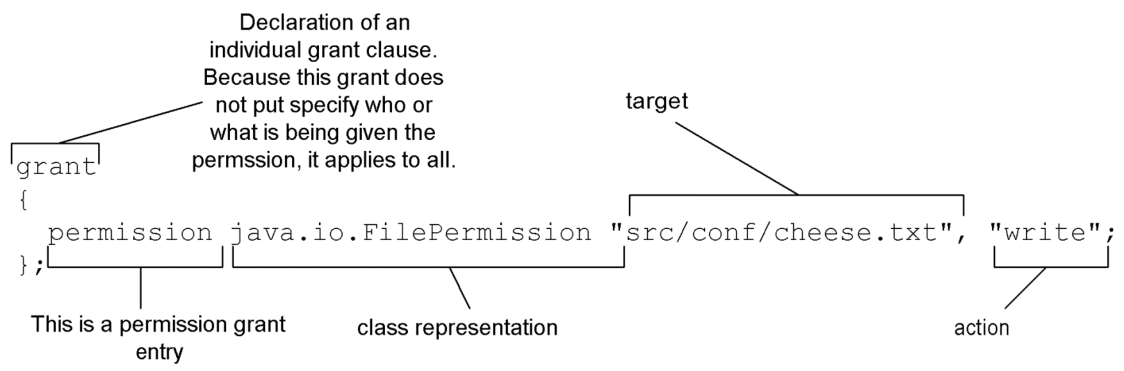
Backing the `SecurityManager` and `AccessController` is a policy that expresses which permission ate granted to a given security context. Policies map the entities that might attempt to access the resources to their `Permissions`. For example, the code found on the `D:` drive may be disallowed from modifying any files on the `C:` drive. As we'll see later in this chapter, this can also be applied to users in addition to code. The main purpose of separating out this concept is that the policy is the logical place for deployment-time configuration. Keeping this responsibility in one place makes managing your security system easier and less error prone.

In the Java security framework, a policy is represented by sub-classes of the abstract class `java.security.Policy`. This class is always a singleton: there is only ever one instance in the VM. Because of this, a `Policy` can be thought of as a service for resolving `Permission` checks. The `Policy` in use can be specified as a VM argument, or can be changed at runtime by calling the static method `Policy.setPolicy(policy)`. Because the `Policy` being used can be swapped out as needed, custom `Policys` can be used in your Java applications. As we'll see, coupled with the generic nature of JAAS classes, this allows you to use the Java security framework as a rich user based permission system.

### The default policy implementation

By default, there is no security manager in effect. This effectively allows all permissions for all resources. If the argument `-Dsecurity.manager=`*`classname`* is passed to the VM at runtime, an instance of *classname* is constructed and used. If the property is supplied with no value, a default implementation is chosen.

If the default security manager is specified, the default policy implementation receives its permissions via a flat-file that can be specified at run-time via the VM argument `-Dsecurity.policy.file=`*`policyfilename`*. If the path is not specified, the file `$JAVA_HOME/lib/security/java.policy` is used.


Declaration of an individual grant clause. Because this grant does not put specify who or what is being given the permssion, it applies to all.
target
This is a permission grant entry
class representation
action

```
grant
{
  permission java.io.FilePermission "src/conf/cheese.txt", "write";
};
```

**Sample grant clause from the configuration file for Java's default policy implementation**

This policy gives all code the ability to write to the file named "`src/conf/cheese.txt`". Without this permission an attempt to modify the file will result in a `SecurityException`.

### *The Default Policy File*

This default policy file is located at `JAVA_HOME/lib/security/java.policy`. This location can be changed with the VM argument `java.security.policy`, which points to the file path of the policy file to use. Also, modifying the `policy.url` properties in the file `JAVA_HOME/lib/security/java.security` will change the location that default policy file is read from.

As we'll see in later chapters, you can also change the policy being used at runtime, even specifying a class to use to resolve permission checks instead of a flat file.

## *1.4. JAAS*

When Java code is executing, it needs to figure out which `Permissions` to apply to the current thread of execution. That is, when JAAS is doing a `Permission` check, it must answer the question "which `Permissions` are currently granted?" As mentioned above, JAAS associates `Permissions` with a "`Subject`."4 In most cases, a `Subject` can be thought of as a "user," but put more broadly, a `Subject` is any entity that Java code is being executed on behalf of. That is, a `Subject` need not always be a person who's, for example, logged into a system with their username and password.

For example, when a person logs into a JAAS-enabled online banking system, the system creates a `Subject` that represents that user. JAAS resolves which `Permissions` the `Subject` has been granted, and associates those `Permissions` with the `Subject`. A "non-human" `Subject` could be another program that is accessing the application. For example, when the nightly batch-processing agent authenticates, or logs into, the system, a `Subject` that represents the agent is created, and the appropriate `Permissions` are associated with that `Subject`.

### *1.4.1. Authenticating Subjects*

So, the first step in JAAS taking effect is logging a user into the system or "authenticating" the users. When a system authenticates a user, it establishes that the user is who they claim to be. As a real world example, when a bank asks one of its clients for their driver's license and compares the picture to them, they're authenticating the client's identity. Similarly, when a user logs in to their bank's online banking application, they're prompted to provide a username and password. Because the user knows both of these items, called "credentials," the online banking system trusts [believes?] the claim that they're Joe User, and allows them to

4 Actually, in the Java security model, `Permissions` can be associated directly with code as well, further specified by the code's origin. For example, you could specify that all classes loaded from the JAR somecode.jar be given a specific set of `Permissions`. This will be covered in later chapters. For our purposes here, we'll skip this detail.

see their account balances, transfer money, and pay bills.

### *1.4.2. Principals: Multiple Identities*

JAAS doesn't directly associate a user's identities with a `Subject`. Instead, each `Subject` holds onto any number of `Principals`. In the simplest sense, a `Principal` is an identity. Thus, a `Subject` can be thought of as a container for all of `Subject`'s identities, similar to how your wallet contains all of your id cards: driver's license, social security, insurance card, or pet store club card. For example, a `Principal` could be:

- The user "jsmith," which is John Smith's login for the server.
- Employee number #4592 which is John Smith's employee number.
- John's Social Security number which is used by the HR department.

Each of these identity `Principals` is associated with John Smith and, thus, once John authenticates with the JAAS-enabled system, each `Principal` is associated to his `Subject`.

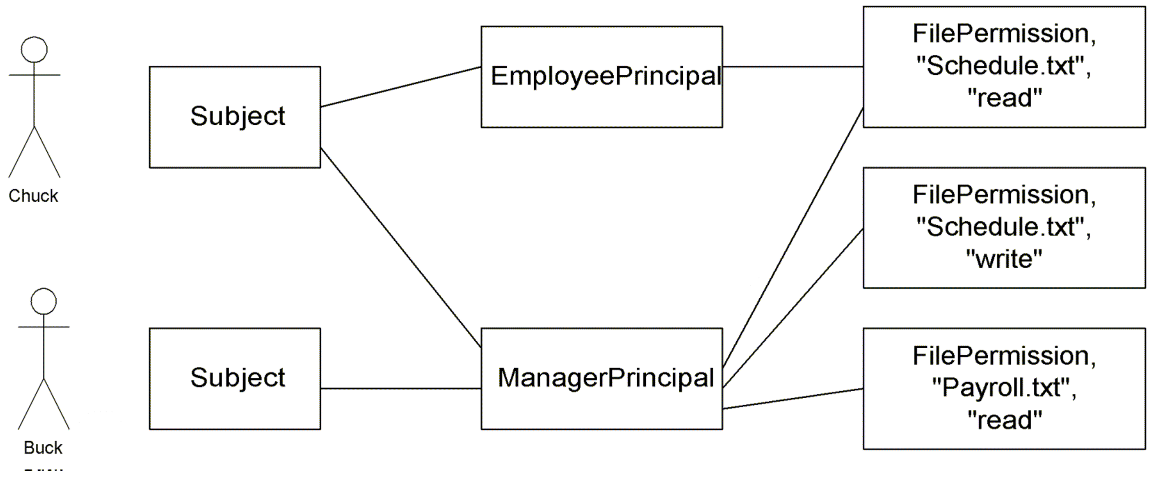
Breaking out a `Subject`'s identities into `Principals` creates clear lines of responsibility between the two: `Principals` tell JAAS who a user is, while `Subjects` aggregate the user's multiple identities. Also, this scheme allows for easier integration with non-JAAS authentication systems, such as single-signon services. For example, when a `Subject` is authenticated with a single-signon service, all of the different users are converted to `Principals`, and bundled into the `Subject`. In this scheme, the `Subject` is an umbrella for all the different identities the user, as represented by a `Subject`, can take on.

### *1.4.3. Subjects and Principals: Roles*

In addition to `Principals` representing different identities of a `Subject`, they can also represent different roles the Subject is authorized to perform. For example, John Smith can perform the following of roles:

- Administer users, including approving new user requests, deleting users, or resetting their passwords.
- Set system-wide configuration properties, like which mail server to use, or the name of the company.

Each of these roles can be encapsulated in JAAS as a `Principal`. Two roles could be created for the above items: User Administrator, System Administrator. As with identities, rather than directly associate each role's abilities, or permissions, with a `Subject`, JAAS associates the role's abilities with `Principals`. John Smith's `Subject`, then, would have `Principals` that represent both of these Administrator roles.


Chuck
Buck
Subject
Subject
EmployeePrincipal
ManagerPrincipal
FilePermission, "Schedule.txt", "read"
FilePermission, "Schedule.txt", "write"
FilePermission, "Payroll.txt", "read"

**A mapping of Employee and Manager roles to permissions. Note that Chuck is an employee AND a manager.**

Separating roles into their own `Principals` has the same dividing-up-responsibility advantages as separating identities. The primary benefit, however, is with managing and maintaining the user's permissions. The management of who can do what in these systems can become extremely cumbersome and time consuming if a user's abilities are directly associated with each Subject instead of a role-based `Principal`, which is associated with n-users.

For example, suppose we have a system with 5 User Administrator. If we followed a model where permissions were associated directly with `Subjects` instead of `Principals`, each `Subject` would be assigned the `Permission` to approve user requests, delete users, and reset their password. During the lifetime of the system, the permissions one of these User Administrators have will change. For example, suppose we decide that resetting a password for a user should only be done by the user: if a User Administrator reset their password, someone other than the user would know the password and might do evil things with that knowledge, or let it slip into the wrong hands. So, we have to modify the `Permission` of each of those 5 User Administrators, and remove the reset password permission. Similarly, we must visit each of the 5 User Administrators if we add a `Permission`.

While this may not seem too onerous a process for 5 users, doing it for more than 5 users can start to be tedious. For example, suppose we had a user base of 10,000 users, 4,032 of which you want to add the new permission "can delete documents." If `Permissions` were directly associated with `Subjects`, you'd have to visit all 4,032 of those users! Instead, if the 4,032 users are all in the "Document Editor" role, as expressed by a `Principal` in JAAS, you need only edit that one role.

## *1.4.3. Credentials*

In addition to `Principals`, `Subjects` also have `Credentials`. The most common types of credential are a username and password pair. When you log in to your email account, for example, you're prompted to enter your username and password.

Credentials can take many forms other than a username and password:

- A single credential, like your password for a voice mail system.
- A physical credential, like a garage door opener to open your garage.
- A digital certificate.
- A mix of physical and keyed-in credentials, like your ATM card and your PIN number.
- Biometric credentials, like your thumbprint or retinal scan.

Put simply, anything you use to prove your identity is a credential.

In JAAS, `Subjects` hold onto two types of credentials: public and private credentials. A username is, in most cases, a public credential: anyone can see your username. A password is, in most cases, a private credential: only the user should know their password. JAAS doesn't specify an interface, or type, for credentials, as any `Object` can be a credential. Thus, determining the semantics of a credential—what that credential "means"—are left up to the code that uses the credential.

## *1.4.4. Principals and the policy*

Once a Subject has been authenticated, having all of its `Principals` associated with it, JAAS uses the `Policy` service to resolve which `Permissions` the `Principals` are granted. A `Policy` is simply a singleton that extends the abstract class `java.security.Policy`. JAAS uses the `Policy`'s `implies()` and `getPermissions()` methods to resolve which `Permissions` a `Subject` has been granted.

The default `Policy` implementation is driven by a flat-file, allowing for declaratively configuring the `Permissions` pre-runtime. Because this implementation is file-driven, however, it's effectively static: once your application starts, you can't cleanly change the file's contents, thus changing `Permissions`.

To provide a more dynamic, runtime `Permission` configuration, you'll need to provide your own `Policy` implementation. You can swap out the `Policy` in effect through a VM argument, or at runtime. As with many other JAAS functions, the currently executing code must have permission to swap out the `Policy`.

## *1.4.5. Access Control: Checking for Permissions*

Once a `Subject` is logged in, and has it's `Principals` associated with it, JAAS can begin to enforce access control. Access control is simply the process of verifying that the `Subject` has been granted any rights required to executing the code. JAAS implements access control by wrapping `Permission` checks around blocks of code. The block of code could be an entire method, or a single line of code. Indeed, for best performance and the finest grain of security control, wrapping the smallest chunk of code possible is the best option.

Because of it's nature, then, access control is often done in JAAS in a "`try/catch`" fashion: attempting to execute the protected code, and then dealing with security exceptions that are thrown due to failed `Permission` checks. `Permission` checking can also be done in a more query-related fashion: before executing a block of code, you can first see if the `Subject` has the appropriate permissions.

Once the `Subject` is authenticated, and the appropriate `Permissions` are loaded, JAAS-enabled code executes securely. Before the code executes, JAAS verifies that the `Subject` has the appropriate `Permissions`, throwing a security exception if the `Subject` does not. This is what is meant by JAAS's claim of being "code centric": the actual code being protected by `Permissions` often does the checking itself.

In most cases, the JAAS model of access control requires the code that is performing the sensitive action to do the permission checking itself. For example, instead of blocking access to a calling `java.io.File's delete()` method, the method itself does the security check. This is usually the safest and quickest approach, as finding all the places that call `delete()` is much more difficult than simply putting access checks in the method itself. In this code-centric approach, the code must include `Permission` checks, and, thus, be knowable of which `Permissions` to check.

In some situations, such a model may be too cumbersome to maintain. Each time a new type of `Permission` is added that's relevant to deleting a file, you must modify the `delete()` method to check for this `Permission`. Strategies that use Dynamic Proxies, declarative meta-information (such as XDoclet or Annotations in J2SE 1.5) or Aspect Oriented Programming, can be used to more easily solve problems like this. Indeed, those and similar strategies can often be used as a cleaner alternative to embedding access control code yourself.

### *1.4.6. Pluggability*

As the above more code-level talk implies, JAAS is a highly pluggable system. What this means is that you can provide functionality to JAAS that wasn't originally shipped with the SDK. It also means that you can change the way in which parts of JAAS work. For example, if the default flat-file based `Policy` doesn't fit your requirements, you can implements and use your own, as later sections in this book will detail.

Unfortunately, as with other high-level frameworks, being pluggable also means there's a bit of code that you'll have to write to customize JAAS to your needs. The good news, however, is that you *can* customize it to your needs.

For example, out of the box, JAAS has no idea how to identity users against your HR system. But, you can write a small amount of code that will do just this. Better, instead of having to conform the HR system to how JAAS works, you can customize JAAS to conform to how the HR system works. This is what "pluggable" means in relation to JAAS: you can add in new functionality that wasn't previously there, and you're not restricted to the original intent, design, and functionality of the framework.

The authentication system is pluggable primarily through providing custom `LoginModule` implementations, while the authorization system is pluggable by providing both custom `java.security.Permission` implementations, and `java.security.Policy` implementations. Chapters X and XX cover creating these custom implementations in great detail.

## *1.5 Looking ahead*

The first part of the book covers the different components of JAAS, going into the above major components and "supporting classes" is much more detail. Included in this part will be an example of creating a database-backed `Policy` and custom `Permissions`.

While the first part has many examples of using these JAAS classes, the second part will provide more in-depth examples of common uses of JAAS, such as logging users in, managing user groups, and creating data-centric `Permissions`.

Finally, the appendixes will go over changes in JAAS in J2SE 5.0, standard J2SE `Permissions`, and a concise reference for configuring JAAS.

## *Summary*

Our first encounter with security in Java began with the need to provide a secure web application for accessing employee information. With that problem at hand, we started exploring the broad topic of Java security, and narrowed down to the Java Authentication and Authorization Service, or JAAS. We introduced JAAS's primary concepts and classes: permissions, policies, and the service layers needed to enforce the granting of permissions. While we dipped our toe into the code-waters of JAAS, our discussion remained fairly high-level so that we could establish the domain needed to dive into the code.

# 2. Two Quick Examples

This chapter provides a two quick examples of how JAAS can be used to provide authentication and authorization. The examples are very simple, using the flat-file based `Policy` implementation provided by Sun Microsystems. Because both examples are simple you can get your feet wet enough to understand the basic concepts and prepare for the more in-depth discussion that follows.

## 2.1. A Simple, Cheesy Example

This example illustrates using of a JAAS policy file to grant permissions to the executing code. Our application will check to see if it's been granted permission to write to a file called cheese.txt. The first time we run the application, permission will be denied because the permission has been commented out in the policy file. Then, we'll uncomment the permission grant in the policy file, giving the code permission to write to the file. Finally, with the correct permission granted, the application will be able to write to `cheese.txt`.

### 2.1.1. The "Application"

Here is the application code:

```
package chp02;

import java.io.File;
import java.io.IOException;

public class Chp02aMain {

  public static void main(String[] args) throws IOException {
    File file = new File("build/conf/cheese.txt");
    try {
      file.canWrite();
      System.out.println("We can write to cheese.txt");
    } catch (SecurityException e) {
      System.out.println("We can NOT write to cheese.txt");
    }
  }
}
```

The above code simply checks to see if we have been granted permission to write to the file `build/conf/cheese.txt`. When we run the application for the first time, we'll turn on the Java security manager by specifying in the system property `java.security.manager`. By default, the security manager is very restrictive in what permissions are granted: only the bare minimum needed to execute the program and check some basic system properties are granted. The default set of permissions does not include access to just any file, such as `cheese.txt`.

## *2.1.2 Running Without Permission*

To run the program for the first time, execute the command `ant run-chp02a`. This Ant command will do the following:

1. Compile the code.
2. "Build" the required configuration files, such as the policy.
3. Execute the command to run the application.

The command that runs the application, which the Ant task executes on your behalf, is:

```
java -cp build/java
-Djava.security.manager
-Djava.security.policy=build/conf/chp02a.policy
chp02.Chp02aMain
```

This command turns on the Java security manager, and specified the policy file to use. The security manager performs permission checks as needed, while the policy file describes the permissions that are granted to executing code and users. We'll learn much more about the security manager and the policy file in upcoming sections and chapters.

When this command is run for the first time, you'll see the following output:

```
run-chp02a:
     [java] We can NOT write to cheese.txt
```

This output indicates that the application has not been granted permission to write to the cheese file. A quick look at the policy file will confirm this:

```
grant
{
//  permission java.io.FilePermission "build/conf/cheese.txt", "write";
};
```

We'll go over the policy file format later, but for now all you need to notice is that permission to write to the cheese file has been commented out with the leading "`//`".

## *2.1.3 Running with Permission*

Before we execute the application again, uncomment the grant by opening the file src/conf/chp02a.policy, and deleting the leading double slashes. After doing this, when we run the ant command ant run-chp02a again, we'll see the below output:

```
run-chp02a:
     [java] We can write to cheese.txt
```

With the permission grant uncommented in the policy file, our code now has been granted permission to write to the file `cheese.txt`.

## *2.2 User Based Authentication and Authorization*

In the second example, we're interested in protecting JAAS itself from being hacked by users of the application it's protecting. For simplicities sake, the example won't protect JAAS from all possible attempts to hack it. Rather, the example will focus on simply protecting access to the `Policy` file. The `Policy` file specifies which permissions logged in users, and the application in general, are granted. Any user that can modify the `Policy` file can potentially grant themselves all permissions, compromising the security of the system. Thus, restricting access to the `Policy` file is very important.

The example system will have two types of users: normal users and systems administrators:

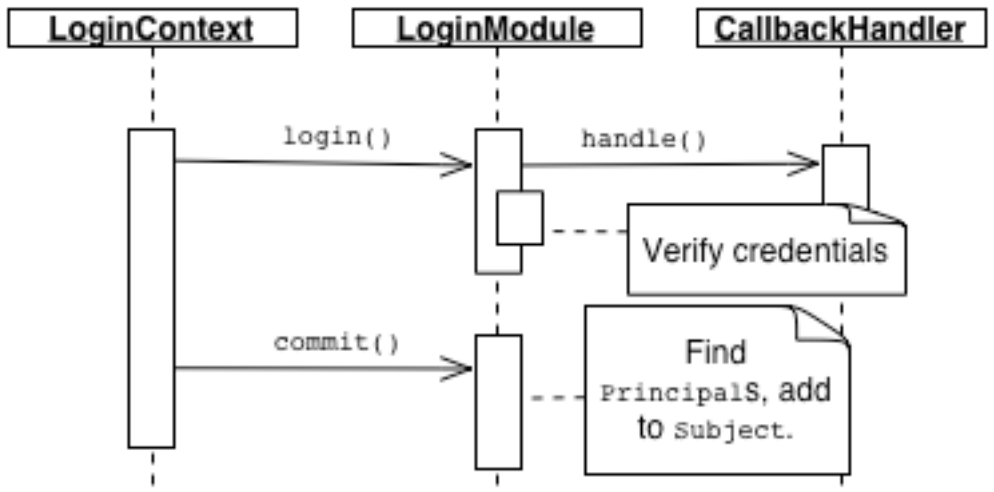
- Normal users cannot access the `Policy` file.
- Only system administrators will be allowed to modify the `Policy` file. Normal users will not.

A `Principal` will represent each of these users. The `Policy` file will declare which permissions each Principal, and thus user, is given. The "application" will be represented by a small class with a `main` method. The application will log each type of user in, and then attempt to access the `Policy` file to demonstrate how access is both checked and restricted with JAAS.

### *2.2.1. Logging in the User*

Before JAAS can be used, the user must be logged in. As noted in the previous chapter, a user is represented by a `Subject`, which holds on to the identities of that user, represented by `Principals`. In the example, then, the concepts of a "normal user" and a "system administrator" are each represented by a `Principal`.

```
UserPrincipal(String username)
SysAdminPrincipal(String username)
```



The diagram below illustrates the high-level process of logging in a user:

1. Collect credentials for the user, done by the `handle()` method on `CallbackHandler`.
2. Verify the credentials, performed by the `LoginModule` implementation.
3. Associated `Principals` accordingly with a `Subject`, also done by the `LoginModule` implementation.

JAAS coordinates all this via the `LoginContext`, which has pluginable items called `LoginModules` that do steps 2 and 3. Multiple `LoginModules` may be configured, allowing multiple sources to contribute `Principals`.

### *2.2.2. The "Application"*

Below is the code that runs the tests for the simple example.

```
package chp02;

import java.io.File;
import java.security.PrivilegedAction;
import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

import chp02.auth.SimpleCallbackHandler;

public class Chp02Main {

  public static void main(String[] args) throws Exception {

    File policyFile = new File("build/conf/chp02.policy");

    testAccess(policyFile, "user", "password");
    testAccess(policyFile, "sysadmin", "password");
  }

  static void testAccess(final File policyFile,
      final String username, final String password)
      throws LoginException {
    //  Login a user
    SimpleCallbackHandler cb = new SimpleCallbackHandler(username,
        password);
    LoginContext ctx = new LoginContext("chp02", cb);
    ctx.login();
    Subject subject = ctx.getSubject();
    System.out.println("Logged in " + subject);

    // Create privileged action block which limits permissions
    // to only the Subject's permissions.
    try {
      Subject.doAsPrivileged(subject, new PrivilegedAction() {

        public Object run() {
          policyFile.canRead();
          System.out.println(username + " can access Policy file.");
          return null;
        }
      }, null);
    } catch (SecurityException e) {
      System.out.println(username + " can NOT access Policy file.");
    }
  }
}
```

The method `testAccess()` is used to test a specific user's ability to read the `Policy` file.

First, a custom `CallbackHandler`, `SimpleCallbackHandler` is instantiated and passed to the `LoginMadule`. `CallbackHandlers` are the part of JAAS that are responsible for collecting the credentials for users. A custom callback handler works in concert with a custom `LoginModule` to authenticate a user, adding Principals to the Subject being authenticated fis all goes well.

The `LoginContext` is a final class in JAAS that coordinates running all the `LoginModules`, and determines what to do if there are any problems along the way. The `LoginContext` is configured through a properties file where each grouping of `LoginModules` are given a name. Thus, when the `LoginContext` is instantiated, the name of the `LoginModule` group is passed to it to tell the `LoginModule` which group to use.

Once the `LoginConext` has authenticated all the users (delegating to the `LoginModules` configured), we can get the authenticated Subject, which will contain the appropriate `Principals`. When the user "user" is authenticated, their `Subject` will have a `UserPrincipal`. When the user "sysadmin" is authenticated, their `Subject` will have a `SysAdminPrincipal`.

Next, with the `Subject` authenticated, we attempt to read the `Policy` file. Creating a `java.io.File` instance for the policy does this. A security check is done within the `canRead()` method, and will throw an exception if it fails.

## *2.2.3. Authentication Code*

There are three custom authentication parts to needed for our example:

1. A custom `LoginModule` for logging in `Subjects`, adding the appropriate `Principals`.
2. A custom `CallbackHandler` to collect a `Subject`'s credentials for our custom `LoginModule`,
3. A configuration properties file to configure JAAS to use the custom `LoginModule`.

In this section, we'll go over each.

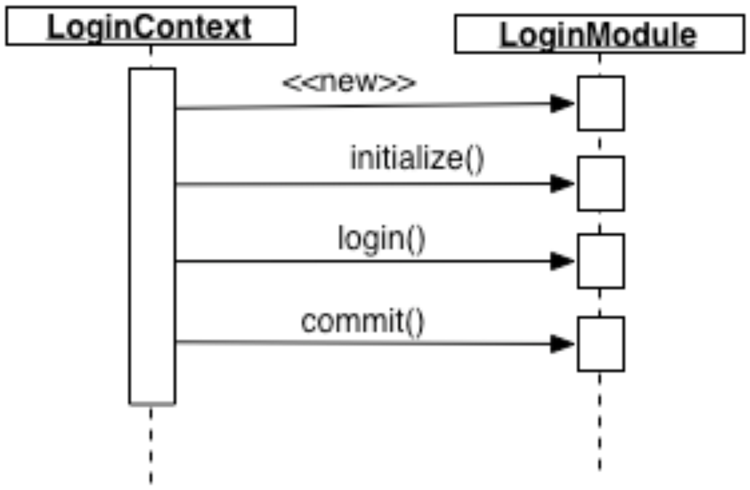
### *Custom LoginModule and CallbackHandler*

`LoginModules` are given the responsibility of authenticating a `Subject` based on the credentials provided. Credentials can be anything that helps confirm the identity of a `Subject`. The most common credentials are username and password. Once a `LoginModule` has verified the identity of a `Subject`, the `LoginModule` will add `Principals`, as appropriate the `Subject`.

JAAS can be configured to use any number of `LoginModules`, allowing disparate authentication sources to contribute `Principals` to a `Subject`. Because multiple `LoginModules` can be used to authenticate a user, a multi-phase process is used to log users in. This is covered in more detail in THE CHAPTER ON LOGINMODULEs. For now, you just need to know that the login module is used the authorize a `Subject`, while the `commit()` method used to add `Principals` to a fully authenticated `Subject`.

The custom `LoginModule` used for the above is below:

```
package chp02.auth;

import java.io.IOException;
import java.security.Principal;
import java.util.Map;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
import chp02.SysAdminPrincipal;
import chp02.UserPrincipal;

public class SimpleLoginModule implements LoginModule {

  private Subject subject;
  private CallbackHandler callbackHandler;
  private String name;
  private String password;

  public void initialize(Subject subject,
      CallbackHandler callbackHandler, Map sharedState, Map options)
  {
    this.subject = subject;
    this.callbackHandler = callbackHandler;
  }

  public boolean login() throws LoginException {
    // Each callback is responsible for collecting a credential
    // needed to authenticate the user.
    NameCallback nameCB = new NameCallback("Username");
    PasswordCallback passwordCB = new PasswordCallback("Password",
        false);
    Callback[] callbacks = new Callback[] { nameCB, passwordCB };
    // Delegate to the provided CallbackHandler to gather the
    // username and password.
    try {
      callbackHandler.handle(callbacks);
    } catch (IOException e) {
      e.printStackTrace();
      LoginException ex = new LoginException(
          "IOException logging in.");
      ex.initCause(e);
      throw ex;
    } catch (UnsupportedCallbackException e) {
      String className = e.getCallback().getClass().getName();
      LoginException ex = new LoginException(className
          + " is not a supported Callback.");
      ex.initCause(e);
      throw ex;
    }

    // Now that the CallbackHandler has gathered the
    // username and password, use them to
    // authenticate the user against the expected passwords.
    name = nameCB.getName();
    password = String.valueOf(passwordCB.getPassword());

    if ("sysadmin".equals(name) && "password".equals(password)) {
      // login in sysadmin
      return true;
    } else if ("user".equals(name) && "password".equals(password)) {
      // login user
      return true;
    } else {
      return false;
    }
  }

  public boolean commit() {
    // If this method is called, the user successfully
    // authenticated, we can add the appropriate
    // Principles to the Subject.
    if ("sysadmin".equals(name)) {
      Principal p = new SysAdminPrincipal(name);

      subject.getPrincipals().add(p);
      password = null;
      return true;
    } else if ("user".equals(name)) {
      Principal p = new UserPrincipal(name);
      subject.getPrincipals().add(p);
      password = null;
      return true;
    } else {
      return false;
    }
  }

  public boolean abort() {
    name = null;
    password = null;
    return true;
  }

  public boolean logout() {
    name = null;
    password = null;
    return true;
  }
}
```



The diagram above illustrates the interaction between the `LoginContext` and `LoginModule` when a `Subject` is being authenticated, the `LoginContext`:

1. Creates an instance of the above `LoginModule`.
2. Calls the `initialize()` method, which gives the `LoginModule` the Subject it will authenticate and the `CallbackHandler` to retrieve credentials with.
3. Calls the `login()` method on the `LoginModule`, telling the `LoginModule` to attempt to authenticate the user.
4. If the `login()` method succeeds, calls the `commit()` method, signaling the `LoginModule` to add Principals to the Subject.
5. If the `login()` method fails or other errors occur, calls the abort method, signaling the `LoginModule` to do any cleanup needed (this is not shown in the diagram above).

## *A Closer Look at login() and commit()*

In our example, the most interesting methods are the `login()` and `commit()` methods. The `login()` method uses the `CallbackHandler` passed in to the initialize method to collect the credentials required. The `SimpleLoginModule` is only interested in the username and a password. A `NameCallback` and `PasswordCallback` instance are created, and passed to the `CallbackHandler`. The `SimpleCallbackHandler` method `handle` (shown below) simply fills in the passed- in `Callbacks`:

```
package chp02.auth;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;

public class SimpleCallbackHandler implements CallbackHandler {

  private String name;
  private String password;

  public SimpleCallbackHandler(String name, String password) {
    this.name = name;
    this.password = password;
  }

  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(name);
      } else if (callback instanceof PasswordCallback) {
        PasswordCallback passwordCB = (PasswordCallback) callback;
        passwordCB.setPassword(password.toCharArray());
      }
    }

  }
}
```

Once the `CallbackHandler` has collected the username and password, they're stored ion the `SimpleLoginModule` instance. Then, the stored credentials are compared against hard-coded values[1]: if each type of user has the correct password, the login method returns true, indicating that the `SimpleLoginModule` has verified the identity of the `Subject`.

The `LoginContext` calls the commit method once the Subject being logged in has been authenticated with all the `LoginModules` required. `SimpleLoginModule`'s commit method is repeated below:

```
public boolean commit() {
    if ("sysadmin".equals(name)) {
      // sysadmin Principal
      Principal p = new SysAdminPrincipal(name);
```

[1] In a real system, of course, the credentials wouldn't be hard-coded; they would be looked up in a database or otherwise retrieved.

```
        subject.getPrincipals().add(p);
        password = null;
        return true;
      } else if ("user".equals(name)) {
        // login user Principal
        Principal p = new UserPrincipal(name);
        subject.getPrincipals().add(p);
        password = null;
        return true;
      } else {
        return false;
      }
    }
```

Since the `Subject` has been authenticated by the login method, the `commit()` method need only check which user has logged in, and add the appropriate `Principal` to the `Subject`. The `commit()` method returns `true` if everything went OK, or `false` if something went wrong.

The Principals `SysAdminPrincipal` and `UserPrincipal` are simple implementations of the `Principal` class. We won't go over them here, but THE CHAPTER/SECTION ON PRINCIPALS covers them in more detail.

## *LoginContext Configuration*

JAAS is configured to use the custom `LoginModule` by specifying it's use in a login module properties file. The file specifies groupings of `LoginModules` by "application." Applications are really just ordered groupings of `LoginModules`, each of which may be required or optional for a `Subject` to be successfully authenticated in the context of that application. These groupings may map to the traditional idea of a software application, or they can just be different groupings.

Our example configuration file contains the below:

```
chp01
 {
      chp01.auth.SimpleLoginModule REQUIRED;
 };
```

This configuration creates an application/group named "chp01." Any `Subject` wishing to be authenticated for that application is required to be successfully authenticated by the `SimpleLoginModule`.

A system property is used to specify the location of the configuration file. In our example, when executing the VM, the following system property is specified

```
-Djava.security.auth.login.config=src/conf/chp01-
```

```
loginmodules.properties
```

Many applications will need to set the `LoginContext` configuration in a more dynamic way, programmatically at runtime. Chapter 4 covers this. Using a flat-file works fine for our example.

## *2.2.4. Authorization Code*

Once the `Subject` has been authenticated, we're ready to attempt to access the `Policy` file, showing off how JAAS performs authorization, or permission, checks. The process is as following:

1. The `Subject` is acquired from the `LoginContext`.
2. The static method `Subject.doAsPrivileged` is used to execute a protected block of code on behalf of the `Subject`.
3. The block of code is implemented by a `PrivilegedAction` implementation.

In addition to the code, you must pass in another VM argument that points to the `Policy` configuration file to configure JAAS's default `Policy`.

### *Priviliged Block of Code: doAsPrivleged*

The method `Subject.idoAsPrivleged` is used to demark that a sensitive block of code be executing on behalf of a given `Subject`. By passing in `null` as the last argument to the `doAsPrivleged` method, we're telling JAAS to execute the `PrivlegedAction` code with *only* the `Permissions` granted to the `Subject`. This means that the `Subject` must contain at least one `Principal` that has been granted the permission to read the `Policy` file.

The inline implementation of `PrivlegedAction` acts as a closure to pass to JAAS. It wraps the code to be executed with the permissions granted to the `Subject`. The method `File.canRead()` contains an authorization check that eventually results in code like the following being called:

```
FilePermission filePerm = new FilePermission("some.policy", "read");
AccessController.checkPermission(filePerm);
```

In the above code, we:

1. Create a `FilePermission` instance that represents the permission to read the file `some.policy`.
2. Use the `AccessController` to see if the `Principal` currently logged in has been granted to required permission.

If the user has been granted permission to read the file, the `checkPermission()` method silently succeeds. Otherwise, if the `Subject` does not have `Permission`, an `AccessControlException` is thrown. Thus, if you want to avoid thrown exceptions from disrupting your application, to check a `Permission` you have to wrap a try/catch block around the sensitive code, and catch any `AccessControlException` that's thrown. If the exception is thrown, the access check has failed.

The `Permissions` granted to each `Principal` are specified in a `Policy` configuration file. This file is used by the default, file based, `Policy` that ships with the SDK. The location of this file is specified by a VM argument.

The two grant entries below are of interest to us[2]:

```
   grant Principal chp02.UserPrincipal "user"
   {
    // not granted anything
   };

   grant Principal chp02.SysAdminPrincipal "sysadmin"
   {
     permission java.io.FilePermission "/Users/cote/dev/jaas-
book/build/conf/chp02.policy", "read";
   };
```

Each of the grant sections above is used to grant (or not grant) `Permissions` to specific permissions. The syntax used is to specify the class of the `Principal`, the name the class will have, and then to list the `Permissions` granted to that `Principal`.

The permission to read the `Policy` file is configured by specifying the class of the `Permission` to grant, the path to the file the `Permission` covers (the target), and the action the `Permission` is for. We've purposefully included the commented- out grant for the `UserPrincipal` to emphasis that the `Principal` doesn't have that grant.

When this policy is applied, only `Subjects` that have a `SysAdminPrincipal` with the name "sysadmin" will be able to read the policy file `chp02.policy`.

## *2.2.5. Running the Example*

To make running the example easier, we've provided an Ant target:

1. Go to the root directory of the source code for this book.
2. Type `ant run-chp02`.

The output will include the following output:

```
run-chp02:
     [java] Logged in Subject:
     [java]     Principal: (UserPrincipal: name=user)
     [java] user can NOT access Policy file.
     [java] Logged in Subject:
     [java]     Principal: (SysAdminPrincipal: name=sysadmin)
     [java] sysadmin can access Policy file.
```

[2] After running the ant target `ant run-chp02`, the policy file will be available at `build/conf/chp02.policy`.

## 2.3. Summary

With the astronaut's- and bird's-eye views of security and Java security, we further brought the discussion down to the worm's-eye view of JAAS in this chapter. By using two simple examples, we discussed the core classes in the JAAS API: as `Policy`, `Permission`, `Subject`, and `Principal`. We discussed the roles of each class and spent time decomposing them into their parts. Without too much detailed discussion, which we've saved for the upcoming chapters, we went over on short example of using JAAS to give you a basic sense of both how JAAS works and what JAAS-enabled code looks like.

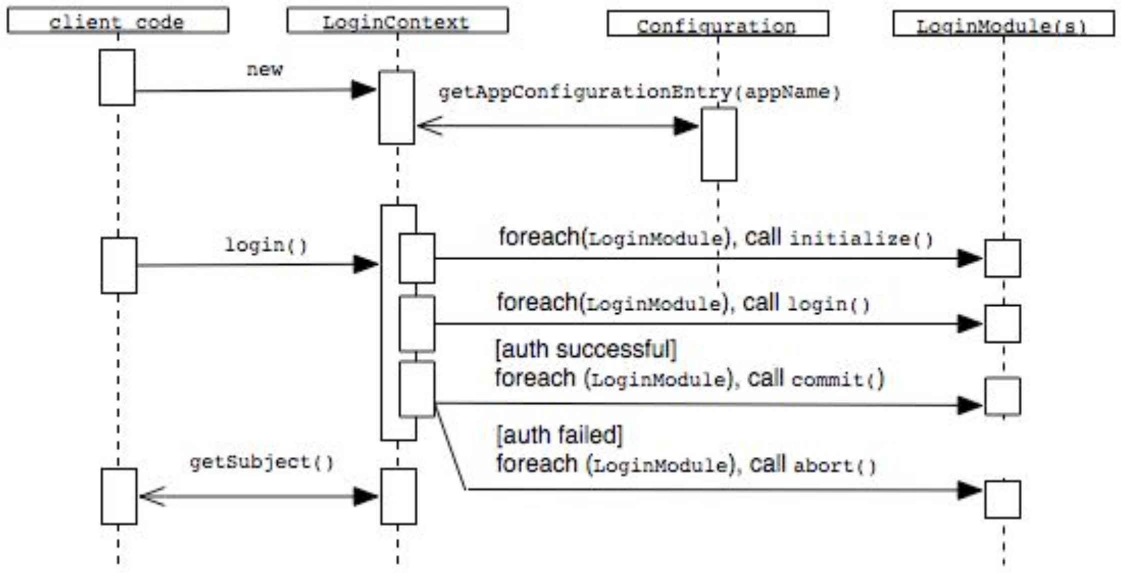
# 3 Authentication

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

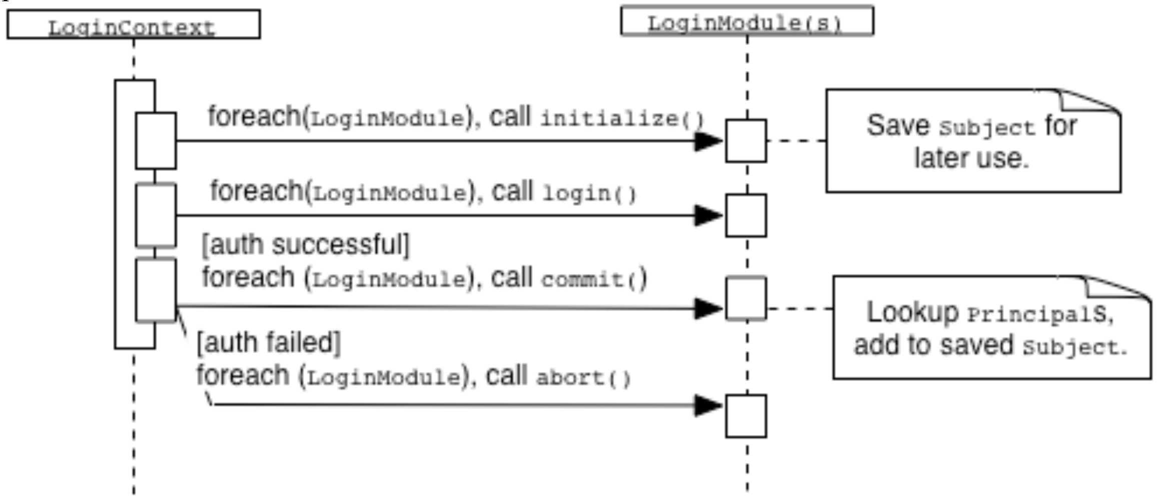
The diagram below illustrates the authentication lifecycle:


client code
LoginContext
Configuration
LoginModule(s)
new
getAppConfigurationEntry(appName)
login()
foreach(LoginModule), call initialize()
foreach(LoginModule), call login()
[auth successful]
foreach (LoginModule), call commit()
[auth failed]
foreach (LoginModule), call abort()
getSubject()

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 arbitrary[1] 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:


LoginContext
LoginModule(s)
foreach(LoginModule), call initialize()
Save Subject for later use.
foreach(LoginModule), call login()
[auth successful]
foreach (LoginModule), call commit()
Lookup Principals, add to saved Subject.
[auth failed]
foreach (LoginModule), call abort()

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 own[2]. 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
```

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.

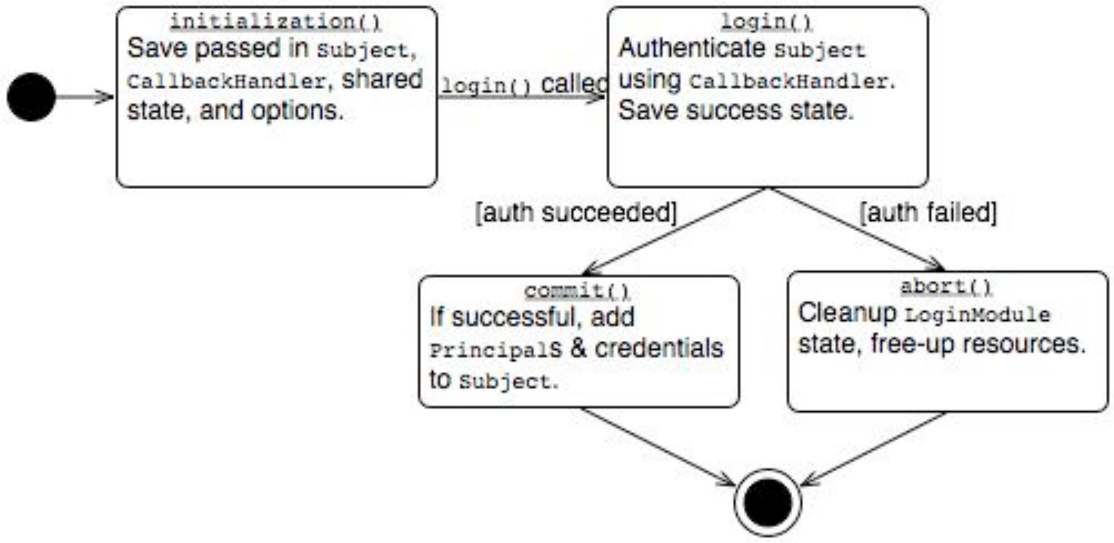
`LoginModules` are "stackable" because multiple `LoginModules` can be used to authenticate one `Subject`. You may need to use more than one LoginModule because your application may have more than identity management system. For example, your application could be an employee records system that needs 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`.

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:


initialization()
Save passed in Subject, CallbackHandler, shared state, and options.
login() called
login()
Authenticate Subject using CallbackHandler. Save success state.
[auth succeeded]
[auth failed]
commit()
If successful, add Principals & credentials to Subject.
abort()
Cleanup LoginModule state, free-up resources.

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 to returned; if it failed, a javax.security.auth.LoginException, or one of it's sub-classes, is thrown; if the LoginModule should be ignored [for what reason?], false is returned. Each LoginModule stores whether is 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.

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 it's `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 is 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",
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-safe[3]. 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 `CallbackHanlder`'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 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 it's `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

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.

### *Verifing 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);
      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 `UserGroupPrincipals` for each, adding them to the `Subject`. Next, the `commit()` method adds a credentials for the username. The class `UsernameCredentual` 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.

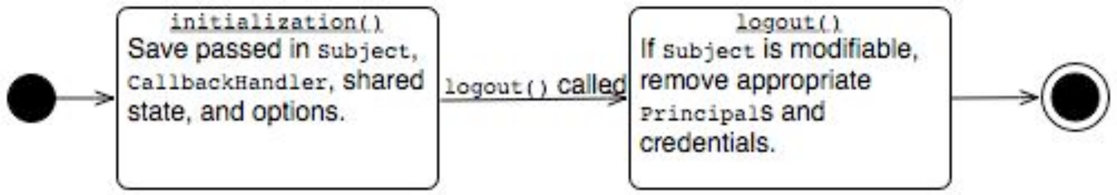
Once the `commit()` method is done, it returns `true` to indicate that everything went well. If an error occurs in `commit()`, a `LoginException` may be thrown. If this `LoginModule` should be ignored, as with the `login()` method, `commit()`should return false.

### *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()*


initialization()
Save passed in Subject, CallbackHandler, shared state, and options.
logout() called
logout()
If Subject is modifiable, remove appropriate Principals and credentials.

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`.

## *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.

# *4 Database-Backed Authentication*

This chapter covers the creation and use of a database-backed `javax.security.auth.spi.LoginModule` and `javax.security.auth.login.Configuration`. The default `Configuration` implementation provided by the J2SE SDK is flat-file based, requiring VM arguments to use. Our dynamic `Configuration` is backed by a database, allowing it be changed during runtime much more easily than through flat-files. The `LoginModule` we'll develop will verify a `Subject`'s credentials against those stored in the database, and then add a `Principal` for each user group the `Subject` is a member of, as specified in the database. As in the previous chapter, each `Subject` will be given a credential that wraps their username.

To allow us to focus on JAAS itself for this chapter, the "application" that uses JAAS will be a simple class with a `main()` method, as in our quick example. The second section of this book will cover integrating JAAS in more real-world applications. For our database, we'll use the easy-to-use, embedded database Hypersonic, with simple JDBC for data access. These two choices don't provide the full-blown data access layer you'd see in a production application, but they allow us to focus on using JAAS itself, rather than the use of production-grade data access layers.

## *4.1 The Application*

Our "application" class, which will drive the examples in this chapter is below:

```
package chp04;

import java.security.Principal;
import java.util.Collections;
import java.util.Iterator;
import java.util.Set;

import javax.security.auth.Subject;
import javax.security.auth.login.AppConfigurationEntry;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

public class Main {

  public static void main(String[] args) throws Exception {
    // set Configuration
    DbConfiguration.init(); #1
    // create DbLoginModule entry
    String appName = "simpleDb";
    DbConfiguration dbConfig = DbConfiguration.getDbConfiguration();

    AppConfigurationEntry appEntry = new AppConfigurationEntry(
        DbLoginModule.class.getName(),
        AppConfigurationEntry.LoginModuleControlFlag.REQUIRED,
        Collections.EMPTY_MAP);
    dbConfig.deleteAppConfigurationEntry(appName, appEntry
        .getLoginModuleName());
    dbConfig.addAppConfigurationEntry(appName, appEntry); #2
    // create LoginContext, login
    String username = "mcote";
    String password = "secret";
    LoginContext ctx = new LoginContext(appName,
        new DbCallbackHandler(username, password));
    // authenticate user
    boolean authenticated = true;
    try {
      ctx.login(); #3
    } catch (LoginException e) {
      e.printStackTrace();
      authenticated = false;
    }
    if (authenticated) {
      // print username
      Subject subject = ctx.getSubject();
      Set creds = subject
          .getPublicCredentials(DbUsernameCredential.class);
      System.out.println("Subject's username: "
          + creds.iterator().next()); #4
      // print principals
      for (Iterator itr = subject.getPrincipals().iterator(); itr
          .hasNext();) {
        Principal p = (Principal) itr.next();
        System.out.println("Principal: " + p.getName()); #5
      }
    } else {
      System.out.println("Did not authenticate " + username);
    }
  }
}
```

(annotation) <#1 Dynamically sets the `Configuration` to use by calling the `DbConfiguration.init()`.>
(annotation <#2 Creates an `AppConfigurationEntry` for `DbLoginModule`, adding it to the database, specifying a name and the `login()` method is required to succeed.>
(annotation) <#3 Uses a `LoginContext` instance to authenticate user mcote.>
(annotation) <#4 Prints out the username credential for the authenticated `Subject`.>
(annotation> <#5 Prints out the `Principals` the login process added to the `Subject`.>

To run the example, make sure you're in the base directory of this book's code, and type `ant run-chp04`. The output of running the above code will include:

```
Subject's username: (DbUsernameCredential: name=mcote)
Principal: sysadmin
Principal: users
Principal: austin-lab
```

Though our example application is overly simple, it represents the process you'll most likely follow in your code. First, you tell JAAS which `Configuration` to use, adding entries for `LoginModules` as needed. Once the `Configuration` is set, you can create a `LoginContext`, passing in a `CallbackHandler` that can acquire credentials as needed. After the `LoginContext` has successfully authenticated the `Subject`, you can access the `Subject`'s newly added `Principals` and credentials with `Subject`'s get methods.

## *4.2 Configuration*

As explained in the previous chapter, the authentication process in JAAS relies on named groups of `LoginModules`. In JAAS, a `Configuration` provides these `LoginModule` groups during the authentication process. A group of `LoginModules` is represented by an array of `AppConfigurationEntry` instances, each of which specifies a `LoginModule` implementation, its control flag, and an optional `Map` of configuration properties.

The abstract class `javax.security.auth.login.Configuration` provides both an interface that all `Configuration` implementations must implement, and static methods for setting and getting the current `Configuration`. Our database-backed implementation, then must, at a minimum, implement the abstract methods `getApplicationConfigurationEntry()` and `refresh()`. In addition to these methods, `DbConfiguration` provides methods for creating and deleting `ApplicationConfiguratioEntrys`.

The full implementation of `chp04.DbConfiguration` implementation is below[1]:

```
package chp04;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Locale;

import javax.security.auth.login.AppConfigurationEntry;
import javax.security.auth.login.Configuration;
import javax.security.auth.login.AppConfigurationEntry.
```

[1] Logging and exception handling code has been removed for brevity.

```
        LoginModuleControlFlag;

import util.db.DbService;

public class DbConfiguration
    extends Configuration {

  static private DbConfiguration dbConfig;

  static public void init() {
    dbConfig = new DbConfiguration();
    Configuration.setConfiguration(dbConfig);
  }

  static public DbConfiguration getDbConfiguration() {
    return dbConfig;
  }

  public void addAppConfigurationEntry(String appName,
      AppConfigurationEntry entry) throws SQLException {
    Connection conn = null;
    try {
      conn = DbService.getInstance().getConnection();
      String sql = "INSERT INTO app_configuration VALUES (?, ?, ?)";
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setString(1, appName);
      pstmt.setString(2, entry.getLoginModuleName());
      pstmt.setString(3, controlFlagString(entry.getControlFlag()));
      pstmt.executeUpdate();
    } finally {
      if (conn != null) {
        conn.close();
      }
    }
  }

  public boolean deleteAllAppEntries(String appName)
      throws SQLException {
    Connection conn = null;
    try {
      conn = DbService.getInstance().getConnection();
      String sql = "DELETE FROM app_configuration WHERE appName=?";
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setString(1, appName);
      return pstmt.executeUpdate() > 0;
    } finally {
      if (conn != null) {
        conn.close();
      }
    }

  }

  public boolean deleteAppConfigurationEntry(String appName,
      String loginModuleName) throws SQLException {
    Connection conn = null;
    try {
      conn = DbService.getInstance().getConnection();
      String sql = "DELETE FROM app_configuration "
          + "WHERE appName=? AND loginModuleClass=?";
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setString(1, appName);
      pstmt.setString(2, loginModuleName);
      return pstmt.executeUpdate() > 0;
    } finally {
      if (conn != null) {
        conn.close();
      }
    }
  }

  public AppConfigurationEntry[] getAppConfigurationEntry(
      String applicationName) {
    if (applicationName == null) {
      throw new NullPointerException(
          "applicationName passed in was null.");
    }

    Connection conn = null;
    try {
      conn = DbService.getInstance().getConnection();
      String sql = "SELECT loginModuleClass, controlFlag "
          + "FROM app_configuration WHERE appName=?";
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setString(1, applicationName);
      ResultSet rs = pstmt.executeQuery();
      List entries = new ArrayList();
      while (rs.next()) {
        String loginModuleClass = rs.getString("loginModuleClass");
        String controlFlagValue = rs.getString("controlFlag");
        AppConfigurationEntry.LoginModuleControlFlag controlFlag =
          resolveControlFlag(controlFlagValue);
        AppConfigurationEntry entry = new AppConfigurationEntry(
            loginModuleClass, controlFlag, new HashMap());
        entries.add(entry);
      }

      return (AppConfigurationEntry[]) entries
          .toArray(new AppConfigurationEntry[entries.size()]);
    } catch (SQLException e) {
      throw new RuntimeException(
          "SQLException retrieving for applicationName="
              + applicationName, e);
    } finally {

      if (conn != null) {
        try {
          conn.close();
        } catch (SQLException e) {
        // Time to panic!
        }
      }
    }
  }

  public void refresh() {
  }

  static String controlFlagString(LoginModuleControlFlag flag) {
    if (LoginModuleControlFlag.REQUIRED.equals(flag)) {
      return "REQUIRED";
    } else if (LoginModuleControlFlag.REQUISITE.equals(flag)) {
      return "REQUISITE";
    } else if (LoginModuleControlFlag.SUFFICIENT.equals(flag)) {
      return "SUFFICIENT";
    } else if (LoginModuleControlFlag.OPTIONAL.equals(flag)) {
      return "OPTIONAL";
    } else {
      // default if unknown
      return "OPTIONAL";
    }
  }

  static LoginModuleControlFlag resolveControlFlag(String name) {
    if (name == null) {
      throw new NullPointerException(
          "control flag name passed in is null.");
    }

    String uppedName = name.toUpperCase(Locale.US);
    if ("REQUIRED".equals(uppedName)) {
      return LoginModuleControlFlag.REQUIRED;
    } else if ("REQUISITE".equals(uppedName)) {
      return LoginModuleControlFlag.REQUISITE;
    } else if ("SUFFICIENT".equals(uppedName)) {
      return LoginModuleControlFlag.SUFFICIENT;
    } else if ("OPTIONAL".equals(uppedName)) {
      return LoginModuleControlFlag.OPTIONAL;
    } else {
      // default if unknown
      return AppConfigurationEntry.LoginModuleControlFlag.OPTIONAL;
    }
  }

}
```

### *4.2.3 Setting the Configuration to Use*

The `init()` method creates a new `DbConfiguration` instance, and uses the static method `Configuruation.setConfiguration()`. This sets the global `LoginModule Configuration`, making all JAAS code running in the VM use our `DbConfiguration`.

An alternative to setting a single `Configuration` is to create a composite `Configuration`. This type of `Configuration` would contain any number of `Configuration` instances, and delegate calls to `getAppConfigurationEntry()` to the list of aggregated `Configurations`.

### *4.2.4 getAppConfigurationEntry()*

`DbConfiguration`'s implementation of `getAppConfigurationEntry()` looks up the `LoginModule` group from the database, creating each `AppConfigurationEntry`, and returns the array of `AppConfigurationEntrys`. No options are used or allowed in this example [should we add them, or is it easy to see how it'd be done?], so the empty `Map` provided by `java.util.Collections` is used.

### *4.2.5 refresh()*

`DbConfiguration`'s `refresh()` method does nothing. Because the persistence store is a database, changes to the `LoginModule` groups are automatically refreshed. If we were to implement a caching scheme, where we didn't have to lookup `AppConfigurationEntrys` each time, we might use the `refresh()` method to blow, and reload, the cache as appropriate.

### *4.2.6 Database Methods*

The other methods in `DbConfiguration` are used to maintain the `LoginModule` group database entries.

### *4.2.7 Control Flags*

Unfortunately, `LoginControlFlag` doesn't provide a good way to get a `String` version of instances suitable for storing in a database or other persistence store. The method `controlFlagString()` is used to create a suitable `String` value for `LoginControlFlag` instances. The corresponding method `resolveControlFlag()` is used to go reconstitute the `String` version of the `LoginModule`'s control flag.

## *4.3 DbLoginModule*

The `DbLoginModule` must implement 5 methods: `initialize()`, `login()`, `commit()`, `abort()`, and `logout()`. Both the `login()` and `commit()` methods lookup the information they need from the database, while the `initialize()` and `abort()` methods store or clear out state. Unlike, for example, Servlets, `DbLoginModules` are used once and thrown away, so state may be saved on them. Indeed, this is the only way to effectively implement a `LoginModule`.

The code for `DbLoginModule` is below:

```
package chp04;

import java.io.IOException;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.security.Principal;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Collections;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;

import util.db.DbService;
import util.id.Id;

public class DbLoginModuleNoLogging implements LoginModule {

  private Subject subject;
  private CallbackHandler callbackHandler;
  private Map sharedState = Collections.EMPTY_MAP;
  private Map options = Collections.EMPTY_MAP;
  private Set principalsAdded;
  private boolean authenticated;
  private String username;
  private Id userId;
  private String password;

  public void initialize(Subject subject,
      CallbackHandler callbackHandler, Map sharedState, Map options) {
    this.subject = subject;
    this.callbackHandler = callbackHandler;
    this.sharedState = sharedState;
    this.options = options;
  }

  public boolean login() throws LoginException {
    NameCallback nameCB = new NameCallback("Username");
    PasswordCallback passwordCB = new PasswordCallback("Password",
        false);
    Callback[] callbacks = new Callback[] { nameCB, passwordCB };
    try {
      callbackHandler.handle(callbacks);
    } catch (IOException e) {
      LoginException ex = new LoginException(
          "IOException logging in.");
      ex.initCause(e);
      throw ex;
    } catch (UnsupportedCallbackException e) {
      String className = e.getCallback().getClass().getName();
      LoginException ex = new LoginException(className
          + " is not a supported Callback.");
      ex.initCause(e);
      throw ex;
    }

    // Authenticate username/password
    username = nameCB.getName();
    password = String.valueOf(passwordCB.getPassword());

    //
    // lookup credentials
    //
    Connection conn = null;
    try {
      conn = DbService.getInstance().getConnection();
      String sql = "SELECT id, password FROM db_user "
          + "WHERE username = ?";
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setString(1, username);
      ResultSet rs = pstmt.executeQuery();
      if (rs.next()) {
        String idStr = rs.getString("id");
        userId = Id.create(idStr);
        String storedPassword = rs.getString("password");
        if (storedPassword == null && password == null) {
          authenticated = true;
        } else if (storedPassword != null
            && storedPassword.equals(password)) {
          authenticated = true;
        } else {
          authenticated = false;
        }
      } else {
        // user does not exist...
        authenticated = false;
      }
    } catch (SQLException e) {
      LoginException ex = new LoginException(
          "SQLException logging in user " + username);
      ex.initCause(e);
      throw ex;
    } finally {
      if (conn != null) {
        try {
          conn.close();
        } catch (SQLException e) {
          // Log exception.
        }
      }
    }
    return authenticated;
  }

  public boolean commit() throws LoginException {
    if (!authenticated) {
      return false;
    }
    // set credential
    DbUsernameCredential cred = new DbUsernameCredential(userId,
        username);
    subject.getPublicCredentials().add(cred);
    // lookup user groups, add to Subject
    Set principals = lookupGroups(username);
    subject.getPrincipals().addAll(principals);
    principalsAdded = new HashSet();
    principalsAdded.addAll(principals);
    return true;
  }

  static private Set lookupGroups(String username)
      throws LoginException {
    Set principals = new HashSet();

    Connection conn = null;
    try {
      conn = DbService.getInstance().getConnection();
      String sql = "SELECT principal.id, principal.name, "
          + "principal.class "
          + "fbhqwFROM db_user, principal_db_user, principal "
          + "WHERE db_user.username = ? "
          + "AND principal_db_user.user_id=db_user.id "
          + "AND principal.id=principal_db_user.principal_id";
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setString(1, username);
      ResultSet rs = pstmt.executeQuery();
      while (rs.next()) {
        String idStr = rs.getString("id");
        Id groupId = Id.create(idStr);
        String groupName = rs.getString("name");
        String className = rs.getString("class");
        if (USR_GRP_CLASS.equals(className)) {
          UserGroupPrincipal grp = new UserGroupPrincipal(groupId,
              groupName);
          principals.add(grp);
        } else {
          Principal p = resolvePrincipal(groupName, className);
          if (p != null) {
            principals.add(p);
          }
        }
      }
    } catch (SQLException e) {
      LoginException ex = new LoginException(
          "SQLException logging in user " + username);
      ex.initCause(e);
      throw ex;
    } finally {
      try {
        if (conn != null) {
          conn.close();
        }
      } catch (SQLException e) {
        // Log exception.
      }
    }
    return principals;
  }

  private static Principal resolvePrincipal(String groupName,
      String className) {
    Exception e = null;
    try {
      Class clazz = Class.forName(className);
      if (Principal.class.isAssignableFrom(clazz)) {
        Constructor c = clazz.getConstructor(STR_ARG);

        return (Principal) c
            .newInstance(new Object[] { groupName });
      }
    } catch (ClassNotFoundException ex) {
      e = ex;
    } catch (SecurityException ex) {
      e = ex;
    } catch (NoSuchMethodException ex) {
      e = ex;
    } catch (IllegalArgumentException ex) {
      e = ex;
    } catch (InstantiationException ex) {
      e = ex;
    } catch (IllegalAccessException ex) {
      e = ex;
    } catch (InvocationTargetException ex) {
      e = ex;
    }
    // Log exception
    return null;
  }

  public boolean abort() {
    username = null;
    password = null;
    authenticated = false;
    return true;
  }

  public boolean logout() throws LoginException {
    //
    // Remove usergroup principals
    //

    if (principalsAdded != null && !principalsAdded.isEmpty()) {
      subject.getPrincipals().removeAll(principalsAdded);
    }
    return true;
  }

  protected boolean isAuthenticated() {
    return authenticated;
  }

  protected Subject getSubject() {
    return subject;
  }

  protected Set getPrincipalsAdded() {
    return principalsAdded;
  }

  protected String getUsername() {
    return username;
  }

  static private final String USR_GRP_CLASS = UserGroupPrincipal.class
      .getName();
  static private final Class[] STR_ARG = new Class[] { String.class };
}
```

### *4.3.1 initialize()*

This method saves the passed in parameters as state on the `DbLoginModule` instances. Saving these parameters, especially the `Subject` and `CallbackHandler` is critical for a `LoginModule` implementation.

### *4.3.2 login()*

The most challenging aspect of `LoginModule` implementations is understanding how `CallbackHandlers` and `Callbacks` are used. `Callbacks` are objects that represent a type of credential that a `LoginModule` requests during authentication. A `CallbackHandler` is an object that knows how to gather certain types of `Callbacks`. `DbLoginModule` is only interested in two credentials: username and password, so it creates instances of the JAAS provided `Callbacks` `NameCallback` and `PasswordCallback`. To make sure the correct `CallbackHandler` is used, when we created the `LoginContext` in the `Main` class, we passed in an instance of `DbCallbackHandler`, which knows how to handle `NameCallbacks` and `PasswordCallbacks`. The code for `DbCallbackHandler` is listed in the next section, XXX.

In our example, we always know that the `CallbackHandler` we'll be given is an instance of `DbCallbackHandler`. However, the interface for `LoginModule` can't guarantee this, so we may get a `CallbackHandler` that can't gather the credentials `DbLoginModule` needed. In such cases, the unknown `CallbackHandler` may throw an `javax.security.login.callback.UnsupportedCallbackException`. Also, if there is an error gathering the credentials, a `java.io.IOException` may be thrown. Throwing an `IOException` seems too narrow in web-centric Java world, but it makes sense from the perspective of a single user application, where gathering credentials may actually involve reading from `System.in` or other input streams.

In either case, how you handle the exceptions is determined by the requirements of your `LoginModule`. Instead of wrapping and re-throwing the exceptions, as we do in the example, you may want to silently fail. If you'd like your `LoginModule` to silently fail when it can't authenticate a user, for whatever reason, then you would catch the exception, perhaps logging it, and simply return `false` from `login()`, telling JAAS to ignore this `LoginModule`.

Once `DbLoginModule` successfully authenticated the `Subject`, member field `authenticated` is set to `true`, allowing other methods to check whether authentication was successful or not. The `login()` method returns `true` once it has successfully authenticated a user. Returning `false` would indicate that this `LoginModule` should be ignored, telling JAAS disregard this `LoginModule` when determining if a `Subject` has been authenticated or not.

## 4.3.3 commit()

The `commit()` method is called if authentication for all needed `LoginModules`, as specified each `LoginModule`'s control flag, in a `LoginContext` succeeded. `DbLoginModule`'s implementation adds a `DbUsernameCredential` to the `Subject`'s public credentials, allowing us to retrieve a `Subject`'s username in the future. Next, `commit()` looks up the `UserGroupPrincipals` for the authenticated `Subject`. `UserGroupPrincipals` is a simple `Principal` implementation that wraps around a single name, the name of the user group. A `Subject` may belong to zero or more user groups. To add `Principals` to a `Subject`, you directly modify the `Set` of `Principals` returned by `Subject`'s `getPrincipals()` method. Our `commit()` method delegates to the private method `lookupGroups()` to retrieve user groups from the database.

## 4.3.4 abort()

`DbLoginModule`'s `abort()` implementation cleans up state that was stored on the `LoginModule` instance. More complicated `abort()` methods might remove any state the `LoginModule` saved in the shared session `Map`.

Because the `Subject` should only be modified by the `login()` method, called when overall authentication was successful, `abort()` implementations shouldn't need to revert any modifications done to the `Subject`'s `Principals` or credentials.

## 4.3.5 logout()

When a `Subject` logs out, or is logged out by the system, the `logout()` method is called. There is no guarantee that the `logout()` method will be called on the same instance of the `LoginModule` used to authenticate a user. Because of this `logout()` cannot depend on stored state in the `LoginModule`: `logout()` can certainly *attempt* to use stored state, but it should have a backup plan if the state is not available, and be cautious to avoid null pointers. While stored state may not be available, the `initialize()` method will have been called, assuring that the `Subject`, `CallbackHandler`, shared session `Map`, and `LoginModule` options `Map` are available.

`DbLoginModule`'s `logout()` method first sees if the `Set` of `Principals` added (updated in the `login()` method) is available. If the `Set` is not available, `logout()` looks up the list of `Principals` again, retrieving the username from the `Subject`'s public credentials. Once the `logout()` method obtains the `Set` of `Principals`, it then attempts to remove all of those `Principals` from the `Subject`.

## 4.4 DbCallbackHandler

The class `chp04.DbCallbackHandler` is an example of a "credential cached" [better phrase] `CallbackHandler`: it doesn't actually collect credentials, they're provided through some other means, outside of the responsibility of the `CallbackHandler`, for example, in a JSP page.

The `handle()` method iterates through the `Callbacks` passed in, and fills in the two types of `Callbacks` it knows how to handle, `NameCallback` and `PasswordCallback`. Though `handle()` can throw an `UnsupportedCallback` if it's given a `Callback` it can't handle, `DbCallbackHandler`'s implementation instead ignores such `Callbacks`. Depending on your needs, you may implement that handling differently.

Below is the code for `DbCallbackHandler`:

```
package chp04;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;

public class DbCallbackHandler implements CallbackHandler {

  private String username;
  private String password;

  public DbCallbackHandler(String username, String password) {
    this.username = username;
    this.password = password;
  }

  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());
      }
    }
  }
}
```

## 4.5 Principals and Credentials

`DbLoginModule` adds two types of objects to a `Subject`: one `UserGroupPrincpal` for each user group that the `Subject` is a member of, and one `UsernameCredential` which wraps the `Subject`'s username.

### 4.5.1 UserGroupPrincipal

As we'll see in chapter XXX, `UserGroupPrincipal` is used to associate `Permissions` with a `Subject`. Furthermore, a `Subject` may have any number of `UserGroupPrincipals`. As with `UserGroupPrincipal`, many `Principal` implementations simply wrap the name of the `Principal`, for example, the user group name. Whenever you create a `Principal` implementation, you should override the `equals()` and `hashCode()` methods as well to ensure that the `Principals` can be stored in `Collection` classes. The `UserGroupPrincipal`, overrides both of these methods, simply keying equality and the hash code value off of the `Principal` name:

```
package chp04;

import java.security.Principal;

import util.id.Id;

public class UserGroupPrincipal implements Principal {

  private Id id;
  private String name;

  public UserGroupPrincipal(Id id, String name) {
    if (id == null) {
      throw new NullPointerException("Id may not be null.");
    }

    if (name == null || name.length() == 0) {
      throw new NullPointerException(
          "User group name may not be empty.");
    }
    this.id = id;
    this.name = name;
  }

  public Id getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public boolean equals(Object obj) {
    if (!(obj instanceof UserGroupPrincipal)) {
      return false;
    }
    UserGroupPrincipal other = (UserGroupPrincipal) obj;
    return getName().equals(other.getName())
        && getId().equals(other.getId());
  }

  public int hashCode() {
    return getName().hashCode() * 27 + getId().hashCode() * 27;
  }
}
```

### 4.5.2 UsernameCredential

Much like `UserGroupPrincipal`, `UserCredential` wraps a `String` value, the `Subject`'s username. Creating a class for the credential is necessary because the one of the only way to retrieve the credential is by using the method `getPublicCredential(Class)` on `Subject`. This method takes the `Class`, or super-class, of the credential you want to retrieve, and returns a `Set` of credentials of that type. So, if you were to put in the `String` of the `Subject`'s username, it would become difficult to distinguish the `String` username from other credentials that were simply added to the `Subject`'s credential `Set` as a `String`.

Since credential implementations will be stored in collections, they too should override `equals()` and `hashCode()`. The code for `UsernameCredential` is below:

```
package chp04;

import util.id.Id;

public class DbUsernameCredential {

  private String name;
  private Id id;

  public DbUsernameCredential(Id id, String name) {
    if (name == null || id == null) {
      throw new NullPointerException(
          "name and/or id may not be null.");
    } else {
      this.name = name;
      this.id = id;
    }
  }

  public Id getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public int hashCode() {
    return getName().hashCode() * 13 + getId().hashCode() * 13;
  }

  public boolean equals(Object obj) {
    if (obj == this) {
      return true;
    }

    if (!(obj instanceof DbUsernameCredential)) {
      return false;
    } else {
      DbUsernameCredential other = (DbUsernameCredential) obj;
      return getName().equals(other.getName())
          && getId().equals(other.getId());
    }
  }

  public String toString() {
    StringBuffer buf = new StringBuffer();
    buf.append("(");
    buf.append("DbUsernameCredential: name=");
    buf.append(getName());
    buf.append(")");
    return buf.toString();
  }
}
```

## *4.6 Database Schema*

Finally, we must define the database tables that back the above database calls. The following tables are used:

```
CREATE TABLE app_configuration
  (
  appName varchar(32) NOT NULL,
  loginModuleClass varchar(255) NOT NULL,
  controlFlag varchar(10),
  PRIMARY KEY ( appname, loginModuleClass )
  );

CREATE TABLE db_user
  (
  username(32) NOT NULL,
  password(32),
  PRIMARY KEY ( username )
  );
```

```
CREATE TABLE db_user_group
  (
  name(32) NOT NULL,
  username(32) NOT NULL
  PRIMARY KEY ( name )
  );
```

## *Summary*

While the previous chapter introduced the domain classes that compose JAAS authentication services, this chapter demonstrated a way to customize those classes to create a database-backed authentication layer. First, we went over one way to store the components of the `Configuration` object in the database, allowing you to more dynamically specify the `LoginModules` required to authenticate users. Next, we covered one way to implement database-backed `Subjects` and their `Principals`. Finally, to perform the actual authentication, we created a custom `LoginModule` that used the database to perform credential verification and, if the `Subject` successfully logged in, add the appropriate database backed `Principals` to the `Subject`.

# 5 Permissions and Access Control

Once a `Subject` is fully authenticated by JAAS, the real work of controlling what an authenticated `Subject` may or may not do can begin. In JAAS, a handful of classes define the core of interfaces and service layer for authorization: `java.security.Permission`, `java.lang.SecurityManager`, `java.security.AccessController`, and `java.security.Policy`. `Permissions` are granted to `Principals`, and determine what actions, on which targets, a `Principal` may perform. The `Policy` is the service used to query for which `Permissions` a `Subject`'s `Principals` have been granted. The `AccessController` verifies that a `Subject`, when on whose behalf code is being executing, has a `Principal` that has been granted the `Permissions` needed to execute that block of code.

This chapter goes over these three core classes, their support classes, use and configuration. The domain discussion in this chapter helps lay the foundation for understanding the custom authorization implementation in the next chapter.

## 5.1 java.security.Permission

A `Permission` encapsulates the granted ability to perform one or more actions, usually to some target. For example, you might have a `java.net.SocketPermission` with the actions “accept” and “connect” for the target `mcote.manning.com:5656`, which would grant the ability to accept connections from and connect to the target hostname and port number. `Permissions` are always granted to and associated with `Principals`, instead of directly with `Subjects`.

The class `java.security.Permission` is abstract, so you always deal with sub-classes. Several sub-classes exist in the SDK, such as:

- `java.security.BasicPermission`, which provides an abstract base implementation for creating other Permissions.
- `java.io.FilePermission` (seen in chapter 2), which governs access to the file system
- `java.util.PropertyPermission`, which governs access to system properties

Permissions that are derived from `BasicPermission` follow a hierarchical naming scheme, and typically support a comma-separated list of actions. Other, more complex, `Permissions` like `java.io.FilePermissions` define their own special syntax.

### 5.1.2 Aspects of a Permission

A `Permission` always has a type, implicit in the actual sub-class of `Permission` that it implements. Each instance of any `Permission` is assigned a name. The semantics of a

`Permission`'s name aren't specified, but sub-classes of `Permission` typically treat the name as the target for the permission, like the hostname and port for the above `SocketPermission`. If the `Permission` does not intrinsically have a target, the name is often descriptive of the broad action granted, for example, the ability to log into a system.

Optionally, a `Permission` can specify actions that are granted. These actions are usually things that can be done to the target, such as accepting or creating connections as in the above `SocketPermission` example. If a `Permission` sub-class has actions, it must implement the `getActions()` method to always return the canonical, `String` representation of the actions. The returned `String` is effectively a marshalling of the actions, and should always be the same for a given set of actions. Using the `SocketPermission`, as an example, the the `getActions()` method would always return the String "`accept,connect`", always comma-separated in the same order.

The interface contract for `java.security.Permission` specifies that all `Permissions` are immutable. To satisfy this contract, setting the name and actions of a `Permission` sub-class is only done when the constructor is called. `Permission` sub-classes, then, should never provide set methods that could be used to mutate the `Permission`'s state, such as the name and actions.

When creating your own custom `Permission` sub-classes, you're not limited to having only a name/target and actions. Though you must have a name, your `Permission` implementations could hold onto any type of other values needed to represent the `Permission`. It's a good idea to add only "data objects" to the `Permission` instead of more action-oriented state like service layers, or any code that performs some action. A `Permission` is an immutable representation of a granted right, so associating objects that can change the state of the `Permission` can easily make your `Permission` mutable.

### *5.1.3 implies(Permission)*

The `implies(Permission)` method on `Permission` is used to answer the question "if a `Principal` has been granted the `Permission` at hand, are they also granted the passed-in `Permission`." If one `Permission` implies another, the `Permissions` are not necessarily equal as determined by the `equals()` method. Rather, the `implies()` method determines if the passed-in `Permission` is a subset of the current `Permission`. This means, of course, that `implies()` will return true for `Permissions` that are equal.

For example, suppose a `Principal` has been granted the following `java.io.FilePermission`, which grants read and write access to any file directly under the `/tmp` directory:

```
FilePermission parent = new FilePermission("/tmp/*", "read, write");
```

When the below permission is passed to the above `FilePermission` instance's `implies()` method, `true` is returned:

```
FilePermission child = new FilePermission("/tmp/log.txt", "read,
write");
```

### *5.1.4 Permission Containers*

JAAS provides several containers for permissions. Each `Permission` container must implement the abstract class `java.security.PermissionsCollection`. Implementations are available either by calling the `newPermissionCollection()` method on some `Permission` sub-classes, by instantiating instances of the `java.security.Permissions`, or by custom implementations of `PermissionCollection`. All containers act as collections for permissions, and provide an `implies()` used to query if at least one of the aggregated `Permissions` imply the passed in `Permission`.

#### *java.security.PermissionsCollection*

`PermissionsCollection` provides an interface for any class whose responsibility is to hold onto a group of `Permissions`. The methods on `PermissionCollection` allow you to add `Permissions`, set the instance as read only, get an `Enumeration` of the `Permissions` in the collection, and query the aggregate `Permissions` with an `implies()` method. The implementation of the `implies()` method may optimize how `Permissions` are looked up.

Aside from custom implementations of `PermissionCollection`, there are two ways to obtain a concrete `PermissionCollection` implementation: by calling `newPermissionCollection()` on a `java.security.Permission` object, or by instantiating a `java.security.Permissions` object. The `PermissionCollection` returned by `newPermissionCollection()` methods are intended to store only one type of `Permission`, for example, `java.io.FilePermissions`. The collection of `Permissions` must be homogenous: each `Permission` in a `PermissionsCollection` has the same type.

If a `Permission`'s `newPermissionCollection()` method returns a `non-null` value, only that `PermissionCollection` can safely be used to store collections of the associated `Permission` type. The `java.security.Permission` class requires that `hashCode()` and `equals()` be implemented, seeming to make it safe to store `Permissions` in collections that rely on those methods, like `java.util.HashSets`. In practice, however, `Permission` implementations are not always "collection-safe." For example, `java.io.FilePermission` bases it's `hashCode()` implementation on only the `FilePermission`'s path, meaning that you may loose `FilePermissions` that have the same path, but different actions, if you store them in a some collections[1].

#### *java.security.Permissions*

When you want to store different types of `Permissions` together, you can use the `PermissionCollection` sub-class, `java.security.Permissions` (notice the "s" at the end). This class aggregates any number of `PermissionCollections`, allowing a heterogeneous collection of `Permissions` to be collected together. `Permissions` provides the same behavior as other `PermissionCollection` sub-classes: setting the collection as

[1] Arguably, this could be considered a bug in `FilePermission`'s `hashCode()` implementation. Bug or not, you'll have to deal with it, which means storing groups of `FilePermission`'s in the `FilePermissionCollection` returned by `FilePermission`'s `newPermissionCollection()` method.

read only, adding new `Permissions`, and using implies() to query if any aggregated `Permission` implies a passed in `Permission`.

The only difference is that `java.security.Permissions` can store different types of `Permissions`, not just one type. As we'll see in the next chapter, `Permissions` is a very useful class for implementing `java.security.Policy`'s methods.

## 5.2 java.security.ProtectionDomain

A `ProtectionDomain` represents a "security context," or frame of execution, in which a permission check is performed. This security context is commonly referred to as a "domain," and can be thought of as a snap-shot of the point at which code is being execution where a permission check is to be performed. A `ProtectionDomain`, can encapsulate two things:

1. The `Principal`(s) executing code.
2. The `java.security.CodeSource` that described where the executing code originates , such as a URL to the JAR from which the class was loaded.

With these items, JAAS is given enough information to check if a `Permission` has been granted to either the specified `Principals`, the `CodeSource`, or a combination of the two. When a `Permission` check is finally done, the `Permission` to check and a `ProtectionDomain` wrapping the above will be passed to the `Policy` in effect. The `Policy` will then determine if the security context represented by the `ProtectionDomain` has been granted the `Permission`. When talking about user-centric, role-based permission systems, this means the Policy will be primarily interested in the `ProtectionDomain`'s `Principals`.

For example, using the quick, simple example from Chapter 2, when the code `File.canRead()` is executed, JAAS creates a new `ProtectionDomain` with the logged in `Subject`'s `Principals` and `chp02.Main`'s `CodeSource`. Eventually, this protection domain is passed to the `Policy` `implies(ProtectionDomain, Permission)` where the `Policy` will determine if the passed in `ProtectionDomain` has been granted the `Permission`.

### 5.2.1 Dynamic vs. Static ProtectionDomains

When a `ProtectionDomain` is created with the constructor that takes a `Principal's` array, the `ProtectionDomain` is known as a "dynamic" protection domain. Before J2SE 1.4, class loaders statically bound `Permissions` to `ProtectionDomains` when their corresponding classes were loaded. This meant that changing permissions during runtime overly difficult: once a class was loaded, the `Permissions` that governed access to its methods and members were effectively set in stone.

Dynamic `ProtectionDomains` were introduced in J2SE 1.4, and made modifying `Permission` grants at runtime much easier. Under the dynamic model, when a `ProtectionDomain`'s implies method is called, it first checks it's own optional list of static `Permissions`, and then delegates to the `Policy` in effect. With this scheme, a dynamic

`Policy`, for example backed by a database, can enforce and modify `Permissions` during runtime. The majority of this book focuses on the use of dynamic `ProtectionDomains`.

### *5.2.2 Principals*

A `ProtectionDomain` may optionally have an array `Principals`, available from the `getPrincipals()` method. The `Principals` in this array are the `Principle` of the `Subject`, if any, in the security context "snap-shot." These Principals will be used to lookup the permissions granted in the `Policy`. When code execution occurs outside of the context of a logged in `Subject`, `getPrincipals()` returns an empty array of `Principals`, assuring that a non-null value is always returned from `getPrincipals()`. The array of `Principals` returned in copy of the `ProtectionDomain`'s `Principals`, so modifications to the returned array will have no effect on the underlying `ProtectionDomain`.

### *5.2.3 java.security.CodeSource*

A `CodeSource` is simply meta-information about the place from which a class was loaded: a JAR, a directory on a file system, or any "location" that can be specified by a URL. This book deals primarily with user-centric, role-based permissions, so we don't discuss or use, `CodeSources` in very much detail. Other JAAS material available, such as Scott Oak's *Java Security*, goes into great depth about `CodeSources`.

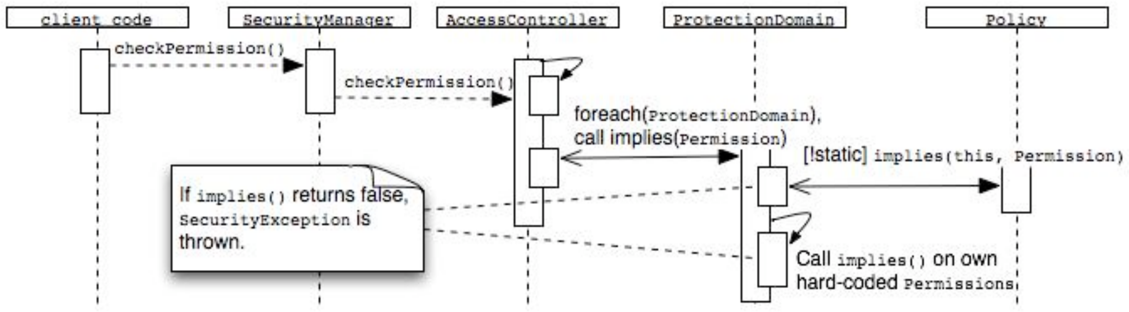
A `ProtectionDomain` can optionally specify what `CodeSource` the code protected comes from, and which digital certificates must have signed the `CodeSource`. A `CodeSource` is simply the URL that the class being granted a `Permission` comes from, and digital certificates used to sign the code. When a class loader loads a class, it remembers the source from which it read the bits for the class, and associates that URL with the `java.lang.Class` instance. `CodeSources` also specify any certificates that were used to sign the class.

In regards to `Permissions`, a `CodeSource` can be thought of a sort of system-level `Subject`. A `Policy` implementation can use `CodeSources` to determine, for example, that code loaded from a remote URL is not allowed to modify any files on the local file system. Indeed, the early Java security models, concerned with providing a secure sandbox to execute applets downloaded from remote sites, relied heavily on this security model.

## *5.3 The SecurityManager*

Since the first version of Java, the class `java.lang.SecurtyManager` provides the service interface for doing all security checks. Earlier versions of Java implemented each permission check by adding a new permission checking method to the `SecurityManager`. This older model explains why there are so many check methods on `SecurityManager`, such as `checkDelete()`, `checkPrintJobAccess()`, or `checkPropertyAccess()`. The SDK 1.2 introduced the `checkPermission(Permission)` method, which each of the above, and other, legacy check methods now delegate to instead of performing their own permission checking. The old check methods are kept for legacy code that calls them directly.

The SecurityManager is enabled by either passing the VM argument java.security.manager, or calling the static method System.setSecurityManager(), passing in the SecurityManager instance to use. Because the SecurityManager to use may be passed into the setSecurityManager(), and because SecurityManager is not a final class, you can provide your own SecurityManager implementation. Before the inclusion of JAAS in the SDK, many application servers, web browsers, and other Java containers did just this to provide an authentication layer. Because there was no standard specified way these implementations should behave however, there was no guarantee that each custom SecurityManager would be implemented in the same way. To provide a standard model of doing authorization, JAAS was introduced.[2] More specifically, java.security.AccessController was made the default security model used by the SecurityManager. Thus, all the check permission methods on SecurityManager eventually delegate to AccessController.checkPermission(). The diagram bellow illustrates the sequence used when SecurityManager's checkPermission() is called:


client code
SecurityManager
AccessController
ProtectionDomain
Policy
checkPermission()
checkPermission()
foreach(ProtectionDomain), call implies(Permission)
[!static] implies(this, Permission)
If implies() returns false, SecurityException is thrown.
Call implies() on own hard-coded Permissions

Though the SecurityManager delegates practically all of it's work to the AccessController, your code should always use the SecurityManager when checking for Permissions, instead of directly calling the AccessController. Doing so ensures that your permission checks will (1) be performed only when security is enabled, and, (2) be performed no matter what SecurityManager is in place.

To obtain the current SecurityManager, you call the static System.getSecurityManager() method, which returns the SecurityManager currently in effect, or null if the SecurityManager is turned "off." Because getSecurityManager() can return null, this leads to the unfortunately necessary convention of always having the check for a null SecurityManager before calling checkPermission(). For example, when checking for permission to read the system property java.version:

```
SecurityManager sm = System.getSecurityManager();
if (sm != null) {
  sm.checkPermission(
    new PropertyPermission("java.version", "read"));
}
```

[2] See *Inside Java 2 Security, 2nd Edition*, pg. 109-112 for more discussion of the history of SecurityManager and AccessController.

If the permission has been granted in the current security context, the `checkPermission()` method silently succeeds. Otherwise, if the `Permission` has not been granted in the current security context, an instance of `java.lang.SecurityException` is thrown.

### *Simplifying Permission Checks*

For convenience sake, to avoid having to create `try/catch` blocks for simple `Permission` checks you may want to use a utility method like the below:

```
static public boolean hasPermission(Permission perm) {
    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
      try {
        sm.checkPermission(perm);
      } catch (SecurityException e) {
        return false;
      }
    }
    return true;
  }
```

The only drawback with such a helper method would be a dependency from your code to the class that contained that helper code, a relatively small price to pay for streamlining the above code.

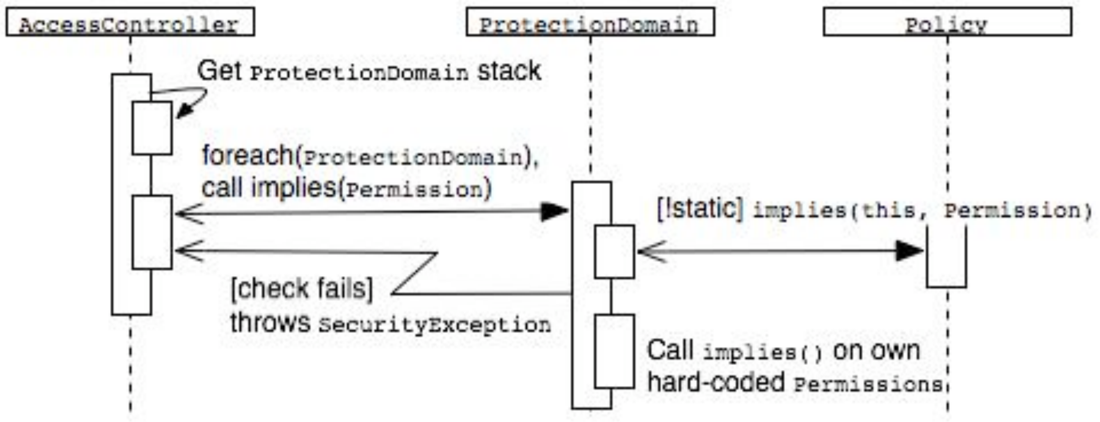
## *5.5 java.security.AccessController*

As it's name implies, the `AccessController` is at the center of JAAS. The `AccessController`'s methods fulfill three responsibilities:

1. Determining if a given `Permission` is granted to the current security context.
2. Executing code in a "privileged" block as needed and allowed, isolating it from complete security checking.
3. Creating security context snap-shots of the current security context to be used in the above two situations.

### *5.5.1 checkPermission()*

The `checkPermission()` is the `AccessController`'s entry point for permission checking. When code needs to perform a permission check, by default, the call to `SecurityManager.checkPermission()` delegates to `AccessController`'s `checkPermission()` method. This method follows the below flow:


AccessController
ProtectionDomain
Policy
Get ProtectionDomain stack
foreach(ProtectionDomain), call implies(Permission)
[!static] implies(this, Permission)
[check fails] throws SecurityException
Call implies() on own hard-coded Permissions

If the `Permission` has been granted to the current security context, `checkPermission()` silently succeeds, returning nothing. If the permission has not been granted, an instance of the runtime exception `java.security.SecurityException` is thrown. This means that, at some level, your code should catch `SecurityException` and attempt to recover accordingly. Methods that contain calls to `checkPermission()` should document which `Permissions` are required, and that the method will throw `SecurityException` if the `Permissions` are not granted to the security context.

As noted in the above discussion of `java.lang.SecurityManager`, to ensure that your code follows the Java security convention and model, the majority of your code should call `SecurityManager.checkPermission()` instead of calling `AccessController.checkPermission()` directly.

## *5.5.2 Privileged Code*

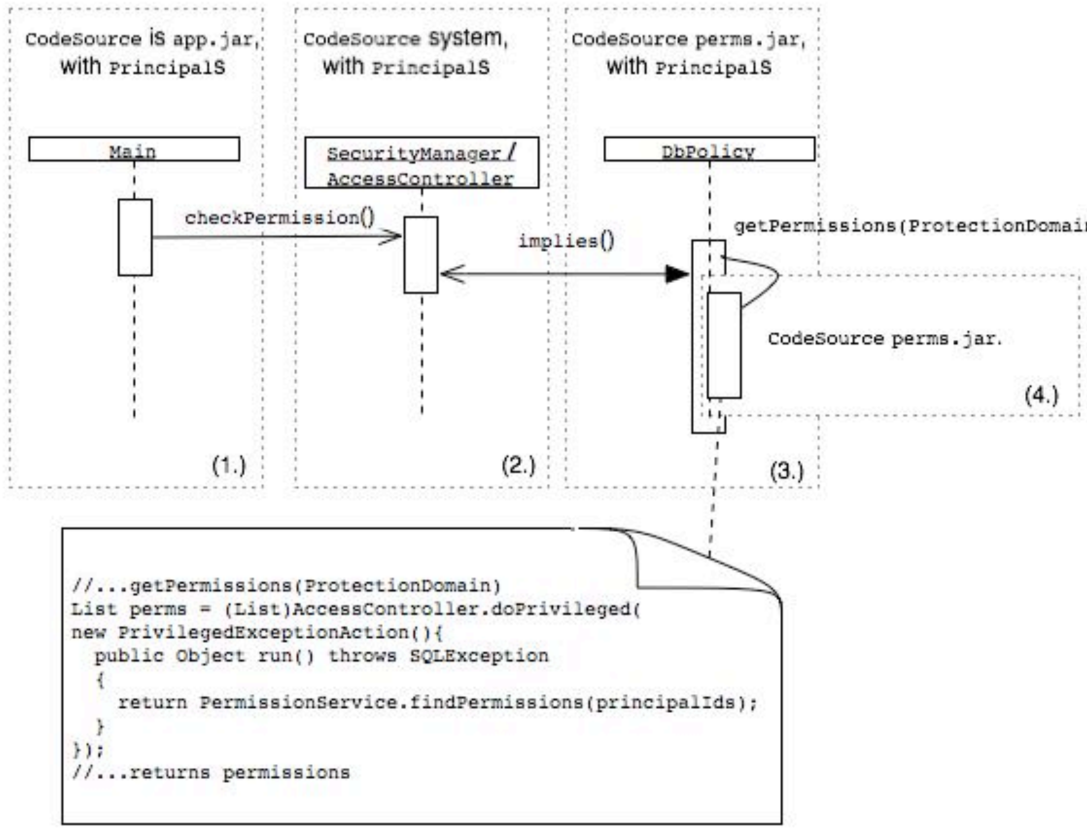
When code is executing, there are times when the security `Policy` currently in effect needs to be ignored. In these cases, the methods `doPrivileged(PrivilegedAction)` and `doPrivileged(PrivilegedExceptionAction)` on `AccessController` can be used to create a privileged security context.

A privileged security context causes a break in the normal security checking. Normally, the `AccessController` calls the `implies()` method for each `ProtectionDomain` in the execution stack, starting from the current code's `ProtectionDomain`. Using a privileged block allows the code that is marked as privileged to perform sensitive operations regardless of the current `Subject`'s granted `Permissions` and the `Permissions` granted to `ProtectionDomains` in the call stack. Instead, only the `ProtectionDomain` of the code marked as privileged is checked.

We'll use our custom `Policy` from the next chapter, `DbPolicy` as an example. When a user is logged in who doesn't have permission to access the database that `DbPolicy`'s information is stored in, the current security context would prevent checking the Db`Policy`. The current security context contains a `ProtectionDomain` for each class in the call stack. Each `ProtectionDomain` contains the `Principals` of the logged in `Subject`, and none of these `Principals` has been granted permission to connect to the database. So, *without* a privileged block, when `DbPolicy` attempts to connect to the database, permission will be denied because none of the `Principals` have been granted the needed permissions.

To fix this problem, two things are done. First, the `DbPolicy`'s `CodeSource`, the JAR `perms.jar`, is granted permission to connect to the database. Second, a privileged block is created when `DbPolicy` looks up the `Permissions` granted to the `Subject`, the code that requires database access. This privileged block prevents the evaluation of the entire `ProtectionDomain` stack (domains 1, 2, 3 in the below diagram), only checking that the code in the privileged block (domain 4 in the below diagram) has been granted permission. `DbPolicy` has been granted the needed permission, so authorization passes, and we can connect to the database.

The diagram below illustrates this example. Each `ProtectionDomain` is represented by a dotted box, and lists the `CodeSource` and whether or not the domain has `Principals`. The note contains the part from the `getPermissions(ProtectionDomain)` method that creates a privileged block.


CodeSource is app.jar, with Principals
CodeSource System, with Principals
CodeSource perms.jar, with Principals
Main
SecurityManager / AccessController
DbPolicy
checkPermission()
implies()
getPermissions(ProtectionDomain)
CodeSource perms.jar.
(4.)
(1.)
(2.)
(3.)

```
//...getPermissions(ProtectionDomain)
List perms = (List)AccessController.doPrivileged(
new PrivilegedExceptionAction(){
  public Object run() throws SQLException
  {
    return PermissionService.findPermissions(principalIds);
  }
});
//...returns permissions
```

## 5.5.3 Creating Security Contexts

The two overloaded versions of the method `doPrivileged()` that take an `AccessControllerContext` as the second argument are used to perform security checks in the security context represented by the passed in `AccessControllerContext`. An `AccessControllerContext` allows you to create a security context to use instead of the current thread's context. One use of this, for example, is to execute code with *only* the `Permissions` of a `Subject`, not those of the system the `Subject` is running in.

### *5.5.4 AccessControllerContext*

An `AcccessControllerContext` instance is used to create a security context, usually one that's different than the currently executing thread. This allows you to create security contexts on the fly, regardless of the permissions the currently logged in `Subject` has been granted. Once an `AccessControllerContext` instance is created, the `checkPermission()` method can be used to query if a permission has been granted in the newly created context. Like many of the classes in the JAAS API, `AccessControllerContexts` are rarely handled directly by the users of the API. Instead, `AccessControllerContext` instances more often used internally within JAAS.

Instances can be created with the two constructors, or by calling `AccessController.getContext()`, which provides a snap shot of the current security context.

## *5.6 SecurityManager vs. AccessController*

While it's still possible to provide and use your own `java.lang.SecurityManager`, it's not advisable, primarily because you would need to devise a new security checking model, or re-invent the wheel, creating the same model that the `AccessController` already provides. Instead, when you want to customize the authorization checks are performed, you should use the default `SecurityManager`, implying the use of `java.security.AccessController`, along with a custom `java.security.Policy`. This strategy provides a ready-to-use design, and an easily pluggable interface that works hand-in-hand with the authentication services provided by JAAS.

## *5.7 Subject.doAs() and Subject.doAsPrivleged*

The `doAs()` methods on `Subject` provide convenience methods for creating security contexts that include the `Permissions` granted to a `Subject`'s `Principals`. The `doAsPrivleged()` methods allows security checks to be done with *only* the `Subject`'s permissions. Additionally, `AccessControllerContexts` can be optionally be passed into `doAsPrivleged()`, allowing further fine-grained control of the security context used.

When `doAs()` or `doAsPrivileged()` is invoked, a `DomainCombiner` is created to add the `Subject`'s `Principals` to each `ProtectionDomain` in the execution stack. These will be the methods you use the most. We'll an example of using `Subject`'s `doAsPrivileged()` in the next chapter, where we implement a custom `java.secuirty.Policy`.

## *5.8 The Policy*

The abstract `Policy` provides the service that answers all queries about dynamic permissions. The `AccessController` delegates permission checks to the `Policy` in effect. For dynamic permission models the `implies(ProtectionDomain, Permission)` method is the central method on `Policy`. This is the method that will be called to resolve if a `Subject`'s

Principals have been granted a Permission. The other methods are either utility methods for maintaining the Policy (setting it, refreshing it), and for supporting legacy code that uses the static permission model.

## *5.8.1 getPermissions(ProtectionDomain)*

The getPermissions(ProtectionDomain) method returns a PermissionCollection of all Permissions granted to the passed in ProtectionDomain. This method is usually used for two purposes:

1. To list all the Permissions a granted to the Principals in a ProtectionDomain, for example, to list them in a page where they're being edited.
2. By the implies() method to lookup the permissions a ProtectionDomain has been granted in order to resolve if a specific permission has been granted.

The default implementation of getPermissions(ProtectionDomain) returns the static Permissions granted to a ProtectionDomain by class loaders and the result of getPermissions(CodeSource) for the ProtectionDomain's CodeSource. Policy implementations that override getPermissions(ProtectionDomain) should maintain this same behavior, in addition to new behavior, for example, looking up a Principal's permissions in a database.

## *5.8.2 implies()*

As with other implies methods, Policy's implies method is used to determine if a security context has been granted a Permission, either directly or indirectly by implication. Policy's implies method takes two arguments: the ProtectionDomain that represents the security context to check, and a Permission. The method returns true if the ProtectionDomain has been granted the passed in Permission, or false if the Permission has not been granted.

The DbPolicy implementation in the next chapter will provide an example of implementing this method.

## *5.8.3 Utility Methods*

```
static getPolicy()
static setPolicy()
abstract refresh()
```

The above utility methods are used to set the Policy implementation to use, and to refresh the Policy currently in effect. Some Policy implementations may not implement any special action when refresh() is called. File-based Policy implementations typically implement the refresh() method to re-read in the file(s) that the Policy uses.

## *Summary*

We've introduced the primary classes that compose JAAS's authorization services. In doing so, we've gone over a detail discussion of JAAS's core authorization classes:

- The permission classes `Permision`, `PermissionCollection` and the heterogeneous `Permissions` container.
- `ProtectionDomain` which is used to describe the permissions granted to a Subject and/or grouping of code.
- The `SecurityManager` and `AccessController` which provide the core services layer for enforcing permission checks. Also, the special `doAs()` methods on Subject that allow you to create `Subject` based access contexts.
- The `Policy`, which provides the service interface for determining which permissions are granted to which `Principals`, and thus, which `Subjects`.

In the next chapter(s) we'll use several of these classes to develop a database-backed dynamic `Policy`.

# 6 A Custom Policy

This chapter describes an example of implementing a custom `java.security.Policy`. The `Policy` we'll develop is actually composed of several parts: a `CompositePolicy` that delegates to any number of "sub-`Policy`" implementations; a database-backed `Policy` that retrieves permission grants from a database; and the default file-based `Policy`. Aggregating a `Policy` like this allows for more flexibility and code reuse. As with previous chapters, we'll use a small class with a `main()` method to demonstrate using the custom `Policy`.

Our custom `Policy` takes one large short cut to simplify the example: `Permission` checks are effectively only applied when a `Subject` is logged in, allowing most code to execute with all permissions enabled. With that disclaimer aside, let's jump into the code.

## 6.1 The "Main" Application

The simple application we use initializes the custom `Policy` and `SecurityManager`, creates a test user, and then does a simple `Permission` check for reading a temporary file. The first check will fail because the required `java.ioFilePermission` hasn't been added to any of the `Subject`'s `Principals`. Before doing a second check, the application associates the needed `FilePermission` to one of the `Subject`'s `Principals`. After the `FilePermission` has been added, the call to `SecurityManager.checkPermission()` returns successfully.

Below is the code used to run the example:

```
package chp06;

import java.io.FilePermission;
import java.security.Policy;
import java.security.PrivilegedAction;
import java.util.ArrayList;
import java.util.List;

import javax.security.auth.Subject;

import util.id.Id;

public class Main {

  static public void main(String[] args) throws Exception {
    AuthHelper authHelper = new AuthHelper();

    try {
      Policy defaultPolicy = Policy.getPolicy();
      DbPolicy dbPolicy = new DbPolicy();
      List policies = new ArrayList(2);
      policies.add(defaultPolicy);
      policies.add(dbPolicy);
      CompositePolicy p = new CompositePolicy(policies);
      Policy.setPolicy(p);

      System.setSecurityManager(new SecurityManager());
      authHelper.createTestUser("testuser", "testpassword");
      authHelper.loginTestUser();
      Subject subject = authHelper.getSubject();

      final FilePermission filePerm = new FilePermission(
          "/tmp/test", "read");

      boolean allowed = true;
      try {
        Subject.doAsPrivileged(subject, new PrivilegedAction() {

          public Object run() {
            SecurityManager sm = System.getSecurityManager();
            if (sm != null) {
              sm.checkPermission(filePerm);
            }
            return null;
          }

        }, null);
      } catch (SecurityException e) {
        allowed = false;
      }

      if (allowed) {
        System.out.println("Subject can read file /tmp/test");
      } else {
        System.out.println("Subject cannot read file /tmp/test");
      }

      Id principalId = authHelper.getUserGrp().getId();
      PermissionService.addPermission(principalId, Id.create(),
          filePerm);
      System.out.println("Added " + filePerm + " to Subject.");

      allowed = true;
      try {
        Subject.doAsPrivileged(subject, new PrivilegedAction() {

          public Object run() {
            SecurityManager sm = System.getSecurityManager();
            if (sm != null) {
              sm.checkPermission(filePerm);
            }
            return null;
          }

        }, null);
      } catch (SecurityException e) {
        allowed = false;
      }

      if (allowed) {
        System.out.println("Subject can read file /tmp/test");
      } else {
        System.out.println("Subject cannot read file /tmp/test");
      }
    } finally {
      if (authHelper != null) {
        authHelper.cleanUp();
      }
    }
  }

}
```

To run the above, change directories to the root of this book's project and execute the command `ant run-chp06`. The output will include:

```
Subject cannot read file /tmp/test
Added (java.io.FilePermission /tmp/test read) to Subject.
Subject can read file /tmp/test
```

## *6.1.1 Initializing the Custom Policy*

As its name suggests, the `CompositePolicy` is an implementation of the Composite pattern: a `CompositePolicy` delegates it's behavior to the 0-to-n `Policys` it holds, and instances of `CompositePolicy` can be used wherever a `java.security.Policy` can be used.

The `CompositePolicy` instance we use is composed of two `Policys`: the default J2SE `Policy`[1], obtained by calling the static method `Policy.getPolicy()` and our custom `chp06.DbPolicy`. The default `Policy` performs Permission assignments and checks for legacy code, and also assigns several basic permissions to every security context. For example, the default Policy grants the permission to access many simple, low-risk properties. Rather than re-implement this behavior, we just re-use the default `Policy` in our `CompositePolicy`. The `DbPolicy`, discussed below in section 6.3, is backed by a database, and provides us with a more dynamic runtime way of doing authorization checks.

Once the `CompositePolicy` and its sub-Policies are created, the application enables the authorization by calling two static set methods: one method to set our `CompositePolicy` as the VM-wide `Policy`, Policy.setPolicy(), and another method to enable the default `java.lang.SecurityManager`, `System.setSecurityManager()`.

The `CompositePolicy` is covered in more detail below, section 6.2.

### *6.1.2 Creating and Authenticating the Test User*

The helper class `chp06.AuthHelper` is used to create, authenticate, and then cleanup the `Subject` and its `Principals` that are used in the example. As the focus of this chapter is authorization, the code isn't excerpted here.

### *6.1.3 Checking Permissions*

To check if the Subject has been authorized to read the file `/tmp/test`, the code first creates a `java.io.FilePermission` instance describing the target and action, creates a security context based on the `Subject`'s authorizations, and then uses the `SecurityManager.checkPermission()`. If the `FilePermission` has been granted, `checkPermission()` silently succeeds, otherwise it throws a `SecurityException`.

The `Subject.doAsPrivileged()` is used to limit the security context to only the `Permissions` granted to the `Subject`'s `Principals`. As discussed in section 5.7 of the previous chapter, the alternative is to use the `Subject.doAs()` method, which instead combines the `Subject`'s `Principals` with the each `ProtectionDomain` in the execution stack, namely the `ProtectionDomain` that represents `chp06.Main`.

## *6.2 The CompositePolicy*

The purpose of `CompostePolicy` is to allow any number of `Policys` to be in effect at one time. As with any `Policy` implementation, the `CompositePolicy` implements the three methods: `getPermissions(CodeSource)`, `getPermissions(ProtectionDomain)`, and the `implies(ProtectionDomain, Permission)`. The first two methods combine the `Permissions` of returned by the aggregated into one `java.security.Permissions` object,

[1] Implemented by the class `sun.security.providers.PolicyFile`, and, by default, backed by the file `<JAVA_HOME/>lib/security/java.policy`.

which is returned. The `implies()` method returns if at least one of the aggregated `Policys` returns true from their `implies()` method; that is, for a `Permission` to be granted to a `ProtectionDomain`, at least one of it's aggregate `Policys` must return true from it's `implies()` method.

The code for `CompositePolicy` is below:

```
package chp06;

import java.security.CodeSource;
import java.security.Permission;
import java.security.PermissionCollection;
import java.security.Permissions;
import java.security.Policy;
import java.security.ProtectionDomain;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public class CompositePolicy
    extends Policy {

  private List policies = Collections.EMPTY_LIST;

  public CompositePolicy(List policies) {
    this.policies = new ArrayList(policies);
  }

  public PermissionCollection getPermissions(ProtectionDomain domain) {
    Permissions perms = new Permissions();
    for (Iterator itr = policies.iterator(); itr.hasNext();) {
      Policy p = (Policy) itr.next();
      PermissionCollection permCol = p.getPermissions(domain);
      for (Enumeration en = permCol.elements(); en
          .hasMoreElements();) {
        Permission p1 = (Permission) en.nextElement();
        perms.add(p1);
      }
    }
    return perms;
  }

  public boolean implies(final ProtectionDomain domain,
      final Permission permission) {
    for (Iterator itr = policies.iterator(); itr.hasNext();) {
      Policy p = (Policy) itr.next();
      if (p.implies(domain, permission)) {
        return true;
      }
    }

    return false;
  }

  public PermissionCollection getPermissions(CodeSource codesource) {
    Permissions perms = new Permissions();
    for (Iterator itr = policies.iterator(); itr.hasNext();) {
      Policy p = (Policy) itr.next();
      PermissionCollection permsCol = p.getPermissions(codesource);
      for (Enumeration en = permsCol.elements(); en
          .hasMoreElements();) {
        Permission p1 = (Permission) en.nextElement();
        perms.add(p1);
      }
    }
    return perms;
  }

  public void refresh() {
    for (Iterator itr = policies.iterator(); itr.hasNext();) {
      Policy p = (Policy) itr.next();
      p.refresh();
    }
  }
}
```

## *6.3 DbPolicy*

The `chp06.DbPolicy` relies on the `UserGroupPrincipal` class introduced in chapter 4, although it supports a wide variety of `Permission` implementations, including it's own `DbPermission`. `DbPolicy` implements the three `Policy` methods, `getPermissions(CodeSource)`, `getPermissions(ProtectionDomain)`, and `implies(ProtectionDomain, Permission)`. As mentioned in the overview, `DbPolicy` takes a shortcut by only providing authorization services for `Subjects`: if the security context does not include a `Subject`, the `DbPolicy` grants all permissions. Though this helps illustrate the inter-workings of JAAS, it can be extremely dangerous depending on your security requirements.

First, we'll look at the code. The rest of this section will then go over different methods in `DbPolicy`.

```
package chp06;

import java.security.AccessController;
import java.security.AllPermission;
import java.security.CodeSource;
import java.security.Permission;
import java.security.PermissionCollection;
import java.security.Permissions;
import java.security.Policy;
import java.security.Principal;
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import java.security.ProtectionDomain;
import java.sql.SQLException;
import java.util.Arrays;
import java.util.Enumeration;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.Set;
import java.util.logging.Level;
import java.util.logging.Logger;

import chp04.UserGroupPrincipal;

public class DbPolicy
    extends Policy {

  public PermissionCollection getPermissions(CodeSource codesource) {
    // others may add to this, so use heterogeneous Permissions
    Permissions perms = new Permissions();
    perms.add(new AllPermission());
    return perms;
  }

  public PermissionCollection getPermissions(
      final ProtectionDomain domain) {
    final Permissions permissions = new Permissions();

    // Look up permissions
    final Set principalIds = new HashSet();
    Principal[] principals = domain.getPrincipals();
    if (principals != null && principals.length > 0) {
      for (int i = 0; i < principals.length; i++) {
        Principal p = principals[i];
        if (p instanceof UserGroupPrincipal) {
          UserGroupPrincipal userGroup = (UserGroupPrincipal) p;
          principalIds.add(userGroup.getId());
        }
      }
      if (!principalIds.isEmpty()) {
        try {
          List perms = (List) AccessController
              .doPrivileged(new PrivilegedExceptionAction() {

                public Object run() throws SQLException {
                  return PermissionService
                      .findPermissions(principalIds);
                }
              });
          for (Iterator itr = perms.iterator(); itr.hasNext();) {
            Permission perm = (Permission) itr.next();
            permissions.add(perm);
          }
        } catch (PrivilegedActionException e) {
          // Log
        }
      }
    } else if (domain.getCodeSource() != null) {

      PermissionCollection codeSrcPerms = getPermissions(domain
          .getCodeSource());
      for (Enumeration en = codeSrcPerms.elements(); en
          .hasMoreElements();) {
        Permission p = (Permission) en.nextElement();
        permissions.add(p);
      }
    }

    return permissions;
  }

  public boolean implies(final ProtectionDomain domain,
      final Permission permission) {
    if (permission.getName().equals("/tmp/test.tx")) {
      int i = 0;
    }
    PermissionCollection perms = getPermissions(domain);

    boolean implies = perms.implies(permission);

    return implies;
  }

  public void refresh() {
    // does nothing for DB.
  }
}
```

## *6.3.1 getPermissions(ProtectionDomain)*

Most of the behavior of `DbPolicy` is done by `getPermssions(ProtectionDomain)`. The other `getPermissions(CodeSource)` method, as noted above, grants all permissions to any `CodeSource`, while the implies method simply uses the `PermissionCollection` returned by `getPermissions(ProtectionDomain)` to perform authorization.

`getPermissions(ProtectionDomain)` first checks if a `Subject` is logged in, by getting the `Principals` associated with the passed in `ProtectionDomain`. If there are no

`Principals`, we assume that there is no `Subject` logged in[2]. In such cases, `getPermissions(ProtectionDomain)` delegates to `getPermissions(CodeSource)`, which grants all permissions.

If a `Subject` with `Principals` is logged in, `getPermissions(ProtectionDomain)` first gathers all of the `Subject`'s `DbUserGroupPrincipal Id`'s, and then uses `PermissionService` to retrieve the associated `Permissions`. The union of all the `Permissions` is returned in a `java.security.Permissions` instance, which allows us to collect different types of `Permissions` together.

### *Privileged Code*

Also, notice that the call to `PermissionService.findPermissions()` is done in a privileged code block. The database code used by `PermissionService.findPermission()` requires that certain `java.io.SocketPermissions` be granted to the current security context. The only `Permission` we've granted to the `Subject`, however, is the ability to read the temporary file `/tmp/test`. The Subject does not have the required `SocketPermissions` to connect to the database.

To get around this, enabling the `DbPolicy` to access the database regardless of which Subject is logged in two conditions must be satisfied:

1. The `ProtectionDomain` that represents `chp06.DbPolicy` must be granted the appropriate `SocketPermissions`.
2. A Privileged code block must be created to execute the sensitive database code in. This Privileged block will exclude the `Subject`'s `Principals` and, thus, `Permission` restrictions, from the security context.

Because of the shortcut we've taken, any code that executes without a Subject is granted `java.security.AllPermission`, granting *all* Permissions, including the `SocketPermissions` we need. So, by creating a Privileged code block around the call the `PermissionService.findPermission()`, we exclude considering the Subject in the authorization checks, thus, allowing the needed `SocketPermission`. See section 5.5.2 in chapter 5 for more discussion on using privileged blocks.

## *6.3.2 UserGroupPrincipals Only*

The `DbPolicy` works only with `UserGroupPrincipals` to support the database schema it relies on. In the database, each `Principal` is assigned a unique `Id`. The Principal interface doesn't guarantee that the `Principal`'s name will be unique in any context, so a different field must be used for a unique identifier, provided by `UserGroupPrincpal`'s `Id` field.

Limiting the `Principals` that a `Policy` supports in this way would be too constrictive if the `DbPolicy` were the only `Policy` that could be used at one time. But, since we provide a `CompositePolicy`, other `Policys` that do not have this limitation can co-exist, allowing other types of `Principals` to be used and supported by other `Policys`. For example, the use

[2] This does leave a loophole in which a `Subject` *with no* `Principals` is granted all `Permissions`.

of the SDK's default `Policy` allows our application to support `Principals` other than `DbUserGroupPrincipal`, albeit through the standard flat-file.

## *6.3.3 The Database and PermissionService*

The `PermissionService` provides the data access layer between the `DbPolicy` and the database. As with the `DbLoginModule`, pure JDBC calls are used to simplify the example. Using JDBC directly like this isn't required, of course. In production systems, it may be appropriate for you to use more featureful data-access libraries like Hibernate, Spring, or EJBs.

`PermissionService` provides methods for adding `Permissions` to `Principals`, looking up the `Permissions` granted to `Principals`, and for removing `Permissions` and their association to `Principals`.

### *Database Schema*

Each of `PermissionService`'s static methods relies, of course, on the permission schema in the database. The schema includes the tables from chapter XXX [authentication chapter], and the tables below:

```
CREATE TABLE permission
  (
  id varchar(64) NOT NULL,
  permissionClass varchar(255) NOT NULL,
  name varchar(64),
  actions varchar(255),
  PRIMARY KEY (id )
  );

CREATE TABLE principal_permission
  (
  principal_id varchar(64) NOT NULL,
  permission_id varchar(64) NOT NULL
  );
```

The `permission` table is responsible for storing each `Permission` assigned to a `Principal`. Each entry must have an `Id` and a fully qualified class name. The name and actions columns are optional. The `principal_permission` table is a tie table used to associate `Permissions` to `Principals`.

### *PermissionService Implementation*

The code for `PermissionService` is listed below. Most of the code is simple JDBC code that does the grunt work of creating, reading, and updating the `Permissions` stored in the database. The primary point of interest are the attempts to reflectively create the loaded `Permissions`, marked by code annotations.

```
package chp06;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.security.Permission;
import java.security.Principal;
import java.security.UnresolvedPermission;
import java.security.cert.Certificate;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
import java.util.Set;
import java.util.logging.Level;
import java.util.logging.Logger;

import util.db.DbService;
import util.id.Id;

public class PermissionServiceNoLogging {

  static private final Certificate[] EMPTY_CERTS = new Certificate[0];

  static private final Class[] ZERO_ARGS = {};

  static private final Object[] ZERO_OBJS = {};

  static private final Class[] ONE_STRING_ARG = { String.class };

  static private final Class[] TWO_STRING_ARGS = { String.class,
      String.class };

  static public void removePermission(Id id) throws SQLException {
    removePermissions(Collections.singleton(id));
  }

  static public void removePermissions(Set ids) throws SQLException {
    Connection conn = null;
    try {
      conn = DbService.getInstance().getConnection();
      String sql = "DELETE FROM principal_permission WHERE
permission_id = ?";
      PreparedStatement tiePstmt = conn.prepareStatement(sql);
      PreparedStatement permPstmt = conn
          .prepareStatement("DELETE FROM permission WHERE id= ?");
      for (Iterator itr = ids.iterator(); itr.hasNext();) {
        // HSQLDB doesn't support addBatch() :(
        Id id = (Id) itr.next();
        tiePstmt.setString(1, id.getId());
        permPstmt.setString(1, id.getId());
        tiePstmt.executeUpdate();
        permPstmt.executeUpdate();
      }
    } finally {
      if (conn != null) {
        conn.close();
      }
    }

  }

  static public List findPermissions(Set principalIds)
      throws SQLException {
    // HSQLDB doesn't allow batching, so we have to do a call per id
    List permissions = new ArrayList();
    for (Iterator itr = principalIds.iterator(); itr.hasNext();) {
      Id principalId = (Id) itr.next();
      permissions.addAll(findPermissions(principalId));
    }
    return permissions;
  }

  static public List findPermissions(Id principalId)
      throws SQLException {
    List perms = new ArrayList();
    Connection conn = null;
    try {
      conn = DbService.getInstance().getConnection();
      String sql = "SELECT permission.id id, "
          + "permission.permissionClass clazz, permission.name name, "
          + "permission.actions actions "
          + "FROM principal_permission, permission "
          + "WHERE principal_permission.principal_id=? "
          + "AND permission.id=principal_permission.permission_id ";

      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setString(1, principalId.getId());
      ResultSet rs = pstmt.executeQuery();
      while (rs.next()) {
        String idStr = rs.getString("id");
        Id id = Id.create(idStr);
        String clazzStr = rs.getString("clazz");
        String name = rs.getString("name");
        String actions = rs.getString("actions");

        Permission perm = null;
        // make class
        Class clazz = null;

        perm = createPermission(id, clazzStr, name, actions);
        if (perm != null) {
          perms.add(perm);
        } else {
          continue;
        }

      }
    } finally {
      if (conn != null) {
        conn.close();
      }
    }
    return perms;
  }

  private static Permission createPermission(Id id,
      String clazzStr, String name, String actions) {
    Permission perm = null;
    Class clazz = null;
    try {
      clazz = Class.forName(clazzStr);
    } catch (ClassNotFoundException e) {
      // deal with below
    }

    if (clazz == null) {
      perm = new UnresolvedPermission(clazzStr, name, actions,
          EMPTY_CERTS);

    } else if (clazz.equals(DbPermission.class)) {
      perm = new DbPermission(id, name, actions);
    } else if (Permission.class.isAssignableFrom(clazz)) {
      try {
        if (name == null && actions == null) {
          Constructor con = clazz.getConstructor(ZERO_ARGS); #1
          perm = (Permission) con.newInstance(ZERO_OBJS);
        } else if (actions == null) {
          Constructor con = clazz.getConstructor(ONE_STRING_ARG); #1
          perm = (Permission) con
              .newInstance(new String[] { name });
        }
        // BasicPermission types
        else if (name != null && actions != null) {
          Constructor con = clazz.getConstructor(TWO_STRING_ARGS); #1
          perm = (Permission) con.newInstance(new String[] { name,
              actions });
        }
      } catch (Exception e) {
        // Log
      }
    }
    return perm;
  }

  static public void addPermission(Id principalId,
      DbPermission dbPermission) throws SQLException {
    addPermission(principalId, dbPermission.getId(), dbPermission);
  }

  static public void addPermission(Id principalId, Id permissionId,
      Permission permission) throws SQLException {
    Connection conn = null;
    try {
      conn = DbService.getInstance().getConnection();
      PreparedStatement pstmt = conn
          .prepareStatement("INSERT INTO permission VALUES (?, ?, ?,
?)");
      pstmt.setString(1, permissionId.getId());
      pstmt.setString(2, permission.getClass().getName());
      pstmt.setString(3, permission.getName());
      pstmt.setString(4, permission.getActions());
      pstmt.executeUpdate();

      PreparedStatement pstmt2 = conn
          .prepareStatement("INSERT INTO principal_permission VALUES
(?, ?)");
      pstmt2.setString(1, principalId.getId());
      pstmt2.setString(2, permissionId.getId());
      pstmt2.executeUpdate();
    } finally {
      if (conn != null) {
        conn.close();
      }
    }
  }

}
```

**(annotation) <#1 the `findPermissions()` method uses reflection extensively to create the `Permission` instances retrieved from the database. Because Permission instances are not required to have a default, no argument constructors, each instance must be reflectively created by attempting to acquire the `java.lang.reflect.Constructor` needed, as dictated by the presence of absence of the name and actions attributes. If an appropriate `Constructor` cannot be found, or and error occurs using it, the Permission is skipped, avoiding the "throwing out the baby with the bathwater" effect where one bad apple ruins the whole barrel. For a much more detailed discussion of reflection, see the Forman's *Java Reflection in Action*.>**

## *Summary*

This chapter demonstrated integrating JAAS authorization functionality with a database. To accomplish this goal, first we created a `CompositePolicy` class that allowed us to use multiple `Policys` at the same time. Next, we created a custom `Policy` implementation that was backed by a database rather than a flat file. Using a database instead of the flat files allows your application to more easily specify `Permissions` at runtime and provides an easier way to maintain all of the `Permission` grants in your system than flat files.

# *7 Authentication Base Classes*

There are several interfaces in JAAS that you can easily end up implementing again and again. In programming, once you perform the same task twice, you should start asking, "what code could I write to prevent having to do this again?" This chapter answers that question with four classes: `BasePrincipal`, `BaseCredential`, `BundleCallbackHandler`, and `ActionsPermission`.

Each class caters to common authentication situations, for example, when a `Subject`'s credentials are username and password, or when a `Principal` can be represented by a simple `String` name.

## *7.1 Base Classes for Principal and credentials*

As we saw in previous chapters, when a `Subject` is being authenticated, two types of classes are encountered over and over again: `Principals` and credentials. `Principals` provide an interface to implement that, in it's most basic form, wraps the `String` name of the `Principal`. Credentials do not provide an interface to implement, except, of course, `java.lang.Object`. However, in many cases a credential will also be a wrapper for a `String`. Here, we provide abstract base classes for both following the model of wrapping `Strings`.

### *7.1.1 BasePrincipal*

`BasePrincipal` is an abstract class that wraps a `java.lang.String` name, provides equals() and `hashCode()` implementations, and is designed to be sub-classed easily. The first motivation behind these features is to support the method `getPrincipals(Class)` on `java.security.Subject`. Principals are not stored as keyed entries (for example, by name), on `Subject`, but are instead "keyed," and grouped together by `java.lang.Class` instances. In this model, the type of a `Principal` becomes part of a `Principal`'s identity. This isn't an extremely common way of identifying data class instances in Java, so it takes a little bit of getting used to. Typically, when you want to identify a particular instance of a class in a collection, such as a `java.util.HashMap`, Java APIs use `Strings`.

`BasePrincipal` supports the model of using the Principals type as part of its identity by including the evaluation of the `Princpal`'s Class in the `hashCode()` and `equals()` method, ensuring that the Principal can be safely stored in collections. By doing this in the base class, concrete implementations of `BasePrincipal` don't need to concern themselves with implementing these two methods. Along with code to support the `getName()` method, implementing the `equals()` and `hashCode()` method satisfies the second motivation behind `BasePrincipal`'s implementation: to provide a quick and easy class to extend when you need a new type of Principal.

The code for `BasePrincipal` is below:

```
package chp07;

import java.security.Principal;

public abstract class BasePrincipal implements Principal,
    Comparable {

  private String name;

  public BasePrincipal(String name) {
    if (name == null) {
      throw new NullPointerException("Name may not be null.");
    }

    this.name = name;
  }

  public String getName() {
    return name;
  }

  public int hashCode() { #1
    return getName().hashCode() * 19 + getClass().hashCode() * 19;
  }

  public boolean equals(Object obj) { #2
    if (this == obj) {
      return true;
    }

    if (!getClass().equals(obj.getClass())) {
      return false;
    }

    BasePrincipal other = (BasePrincipal) obj;

    if (!getName().equals(other.getName())) {
      return false;
    }

    return true;
  }

  public String toString() { #4
    StringBuffer buf = new StringBuffer();
    buf.append("(");
    buf.append(getClass().getName());
    buf.append(": name=");
    buf.append(getName());
    buf.append(")");
    return buf.toString();
  }

  public int compareTo(Object obj) { #3
    BasePrincipal other = (BasePrincipal) obj;
    int classComp = getClass().getName().compareTo(
        other.getClass().getName());
    if (classComp == 0) {
      return getName().compareTo(other.getName());
    } else {
      return classComp;
    }
  }

}
```

**(annotation) <#1 The `hashCode()` method bases it's value on both the Principal name and the class of the `Principal`. To get the class, `BasePrincipal` uses the `getClass()` method to dynamically retrieve the type of the instance, instead of statically referencing `BasePrincipal.class`. This ensures that we use the `java.lang.Class` of the concrete implementation of `BasePrincipal`.>**
**(annotation) <#2 As with `hashCode()`, the equals method uses both the `Principal`'s name, and the `java.lang.Class` obtained by calling `getClass()`, ensuring that sub-classes of the same type will be equal to each other.>**
**(annotation) <#3 We've implemented the `java.lang.Comparable` interface largely for easing testing. For example, if you're using JUnit's `assertEquals()` to compare two collections of `BasePrincipals` (a collection of the `Principals` you're expecting, and a collection your test code returned), JUnit will display the String values of those collections when the assert fails. The order that `BasePrinciapls` listed in will not always be guaranteed, and may very well be different between the two collections. This makes eyeballing an assertion error from JUnit difficult when you're trying to figure out how the two collections are unequal. So, by implementing `Comparable`, depending on the `java.util.Collection` implementation you're using, `BasePrincipal` makes this task easier: the `BasePrincipals` in each collection will be in the same order, allowing you to more easily spot which `BasePrincipals` are different between the two collections.>**
**(annotation) <#4 As with implementing Comparable, the motivation behind implementing `toString()` is primarily to ease testing. Without implementing `toString()`, making sure to display the relevant values of `BasePrincipal`'s name and Class, you would have to use a debugger to evaluate the state of a `BasePrincipal`.>**

### *Example*

A Principal commonly represents a user group, a grouping of users such as "Administrators," "Accounting Department," or other classification of users. In its simplest form, a user group can be represented by the name of the group. In JAAS, in addition to this, the user group would have a type. The implementation of this using `BasePrincipal` as the super class might be like the code below:

```
package chp07;

public class UserGroupPrincipal
    extends BasePrincipal {

  public UserGroupPrincipal(String name) { #1
    super(name);
  }
}
```

(annotation) <#1 implementing **`BasePrincipal`** requires just providing a constructor that takes the name of the **`Principal`**, delegating to the **`BasePrincipal`**'s constructor.>

Next, in a `LoginModule`'s `commit()` method, you would use code like this to associated `UserGroupPrincipals` to the Subject being authenticated:

```
public void commit() {
    if (authenticated) {
      List groupNames = findGroups(userId);
      for (Iterator itr = groupNames.iterator(); itr.hasNext();) {
        String groupName = (String) itr.next();
        UserGroupPrincipal up = new UserGroupPrincipal(groupName);
        subject.getPrincipals().add(up);
      }
    }
  }
```

Once the Subject has been authenticated, you can access the `Subject`'s `UserGroupPrincipals` as the code below demonstrates:

```
LoginContext ctx = new LoginContext("example",
        new BundleCallbackHandler("mcote", "thepassword"));
ctx.login();
Subject subj = ctx.getSubject();
Set groups = subj.getPrincipals(UserGroupPrincipal.class);
```

The items in `groups` will all be of type `UserGroupPrincipal`.

## *7.1.2 BaseCredential*

JAAS doesn't provide an interface or other class that credentials must implement. Instead, as explained in chapter 5, section XXX, credentials are allowed to be of any type. In many instances, credentials can be represented, or are, simple `Strings`. For example, one of the most common credentials, a username, is usually a `String` typed in for a user. `BaseCredential` provides an abstract class for these types of credentials, those that can be represented by a `String`.

The abstract `BaseCredential` is almost identical to the `BasePrincipal` class. Following our original motivation to re-use code, if the code for both is so much alike we might at first think that `BaseCredential` and `BasePrincipal` should be the same class. However, a credential is not a `Principal`, and a `Principal` is not a credential. If both classes were derived from the same base class, they would implicitly be the same thing, if only abstractly. This can lead to confusion when the classes are used, and even programmatic errors where an instance of a credential is used when a `Principal` should be. So, though the code in the `BaseCredential` class is very similar to the code in `BasePrinciapl`, we divide the two concepts into separate classes.

The code is below:

```
package chp07;

public abstract class BaseCredential implements Comparable {

  private String value;

  public BaseCredential(String name) {
    if (name == null) {
      throw new NullPointerException("Name may not be null.");
    }

    value = name;
  }

  public String getValue() {
    return value;
  }

  public int hashCode() {
    return value.hashCode() * 29 + getClass().hashCode() * 29;
  }

  public boolean equals(Object obj) {
    if (this == obj) {
      return true;
    }

    if (!getClass().equals(obj.getClass())) {
      return false;
    }

    BasePrincipal other = (BasePrincipal) obj;

    if (!getValue().equals(other.getName())) {
      return false;
    }

    return true;
  }

  public String toString() {
    StringBuffer buf = new StringBuffer();
    buf.append("(");
    buf.append(getClass().getName());
    buf.append(": value=");
    buf.append(getValue());
    buf.append(")");
    return buf.toString();
  }
  public int compareTo(Object obj) {
    BaseCredential other = (BaseCredential) obj;
    int classComp = getClass().getName().compareTo(
        other.getClass().getName());
    if (classComp == 0) {
      return getValue().compareTo(other.getValue());
    } else {
      return classComp;
    }
  }

}
```

### *Example*

A username is commonly stored as a credential. Like a `UserGroup`, because a String can easily represent a username, using `BaseCredential` is a natural fit for implementing a `UsernameCredential`. As with UserGroupPrincipal and BasePrincipal, the only code required to implement BaseCredential is a one-argument constructor that delegates to the super class:

```
package chp07;

public class UsernameCredential
    extends BaseCredential {

  public UsernameCredential(String name) {
    super(name);
  }

}
```

In a `LoginModule`'s `commit()` method, where the instance field `username_` was the name the user entered, you might use code like the following to add a `UsernameCredential` to the `Subject`:

```
public void commit() {
    //...other code to add Princiapls...
    if (authenticated) {
      UsernameCredential cred = new UsernameCredential(username);
    }
  }
```

Once the `Subject` has been authenticated, you can access the `UsernameCredential` as shown below:

```
LoginContext ctx = new LoginContext("exampleApp",
        new BundleCallbackHandler("mcote", "thepassword"));
    ctx.login();
    Subject subject = ctx.getSubject();
    Set usernames = subject.getPrincipals(UsernameCredential.class);
    if (usernames.size() > 1) { #1
      LOGGER
          .warning("More than one UsernameCredential found.");
    } else {
      UsernameCredential username = (UsernameCredential) usernames
          .iterator().next();
    }
```
**(annotation) <#1 In most cases, a `Subject` will have only one username, so we log a warning message if more than one `UsernameCredential` is found. In cases where there is more than one username, it's generally a good idea to create a different type of `UsernameCredential` for each username so that your code can distinguish between the different usernames.>**

## *7.2 BundleCallbackHandler*

JAAS's original model of gathering credentials is for a `javax.security.callback.CallbackHandler` to gather credentials directly from the user, for example, popping up a Swing dialog box to get a user's username and password. In web applications, this original model of gathering credentials doesn't quite work out so elegantly. Instead, the `CallbackHandler` must know the values of the required credentials ahead of time, caching the values until one of the Callbacks passed into the `handle()` method requests them. We saw this strategy in section 4.3 with our custom `LoginModule`, and in several other sections where a `CallbackHandler` was used.

The most common types of credentials we've encountered are usernames and passwords. Indeed, they're so widely used that JAAS provides two Callback for these credentials out of the box: `javax.security.callback.NameCallback` and `javax.security.callback.PasswordCallback`. To make gathering these, and other, credentials easier, the `BundleCallbackHandler` provides handling for `NameCallback` and `PasswordCallback`, and allows for easily extension to add other Callback types. Set methods are provided to allow a `BundleCallbackHandler` easily cache the credential values ahead of time.

The protected `handleCallback()` method gives sub-classes the opportunity to override `BundleCallbackHandler`'s behavior, re-doing how `NameCallback` and `PasswordCallback` are resolved, or also adding handling for new `Callbacks`. `handleCallback()` returns a `boolean` value indicating if the passed in `Callback` was successfully handled. This return value allows sub-classes to delegate to the super class's implementation, and only attempt to fill out the `Callback` if the super-class wasn't able to.

First, we'll take a look at the code, below. Then, we'll look at an example of extending `BundleCallbackHandler` to handle other types of `Callbacks`. Finally, we'll see an example of `BundleCallbackHandler` in action.

```
package chp07;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;

public class BundleCallbackHandler implements CallbackHandler {

  private static final char[] EMPTY_CHARS = new char[0];
  private Map callbackValues = new HashMap();

  public BundleCallbackHandler() {
  }

  public BundleCallbackHandler(String username, String password) {
    setName(username);
    setPassword(password);
  }

  public void handle(Callback[] callbacks) throws IOException,
      UnsupportedCallbackException {
    if (callbacks == null || callbacks.length == 0) {
      return;
    }
    for (int i = 0; i < callbacks.length; i++) {
      Callback c = callbacks[i];
      handleCallback(c);
    }

  }

  protected boolean handleCallback(Callback callback) {
    if (callback instanceof NameCallback) {
      NameCallback c = (NameCallback) callback;
      if (callbackValues.containsKey(NameCallback.class)) {
        String name = (String) callbackValues
            .get(NameCallback.class);
        c.setName(name);
        return true;
      } else {
        return false;
      }
    } else if (callback instanceof PasswordCallback) {
      PasswordCallback c = (PasswordCallback) callback;

      if (callbackValues.containsKey(PasswordCallback.class)) {
        String password = (String) callbackValues
            .get(PasswordCallback.class);
        if (password == null) {
          c.setPassword(EMPTY_CHARS);
        } else {
          c.setPassword(password.toCharArray());
        }
        return true;
      } else {
        return false;
      }
    } else {
      return false;
    }
  }

  public void setName(String name) {
    callbackValues.put(NameCallback.class, name);
  }

  public void setPassword(String password) {
    callbackValues.put(PasswordCallback.class, password);
  }

}
```

## *7.2.1 Extending BundleCallbackHandler*

When gathering credentials for Windows login, in addition to the username and password, you need the Windows domain. This text `String` can be embedded in the username, but it's more user friendly to gather it as a separate credentials. JAAS provides `javax.security.auth.callback.TextInputCallback` whose purpose is to gather "generic text information," such as a Windows domain. Let's say we want to add the ability to handle `TextInputCallback` to `BundleCallbackHandler`. First, we would create a new class, `WindowsCallbackHandler` that extends `BundleCallbackHandler`. Then, `WindowsCallbackHandler` would override `handleCallback`, delegating to the `BundleCallbackHandler`'s implementation, and then attempting to handle any Callbacks not handled by the super class's method.

The code is below:

```
package chp07;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.TextInputCallback;

public class WindowsCallbackHandler
    extends BundleCallbackHandler {

  private String domain;

  public WindowsCallbackHandler(String username, String password,
      String domain) {
    super(username, password);
    this.domain = domain;
  }
  protected boolean handleCallback(Callback callback) {
    if (!super.handleCallback(callback)) {
      if (callback instanceof TextInputCallback) {
        TextInputCallback c = (TextInputCallback) callback;
        c.setText(domain);
        return true;
      }

      return false;
    }

    return false;
  }

}
```

## *7.2.2 Example of using BundleCallbackHandler*

You use the `BundleCallbackHandler` just as you would any other `Callbackhandler`, passing in an instance of `BundleCallbackHandler` to the `LoginContext` constructor:

```
BundleCallbackHandler bundle = new BundleCallbackHandler("mcote",
  "thepassword");
LoginContext ctx = new LoginContext("exampleApp", bundle);
ctx.login();
Subject subject = ctx.getSubject();
```

`LoginModules` used with `BundleCallbackHandler` instances behave normally. For example, the below is an example of a `LoginModule`'s `login()` method:

```
public boolean login() throws LoginException {
    NameCallback name = new NameCallback("Username:");
    PasswordCallback password = new PasswordCallback("Password:",
        false);
    try {
      callbackHandler.handle(new Callback[] { name, password });
    } catch (IOException e) {
      throw new LoginException(e.getMessage());
    } catch (UnsupportedCallbackException e) {
      throw new LoginException(e.getMessage());
    }
    String username = name.getName();
    String pw = String.valueOf(password.getPassword());
    authenticated = checkPassword(username, pw);
    if (authenticated) {
      return true;
    } else {
      throw new LoginException("User " + name
          + " not authenticated.");
    }
}
```

## 7.3 Base java.security.Permission Classes

Creating a custom permission in JAAS requires you to implement the abstract `java.security.Permission` class, meaning that your new class must provide implementations for `Permission`'s four abstract methods `equals()`, `getActions()`, `hashCode()`, and `implies()`. The SDK provides a base `Permission` implementation for action-less permissions. Additionally, this section provides a base `Permission` implementation for permissions that use actions.

### 7.3.1 java.security.BasicPermission

If the permission you're creating will not use the actions property, known as a "named permission," you can extend the abstract `java.security.BasicPermission` provided in the SDK. `BasicPermission` implements the four abstract methods, providing a no-op method for `getActions()`. Because `getActions()` is ignored, any actions passed into the constructor for a `BasicPermission` will be ignored, and `getActions()` always returns an empty `String`.

`BasicPermission` also provides special handling for the permissions name. The name is treated as hierarchical name, where each level is separated by a period. A wild-card can be used to represent "anything below this level." For example, the class `java.util.PropertyPermission` uses this naming convention to control access to VM properties. One group of properties is the OS group, containing properties such as `os.name` and `os.version`, which store the name and version number of the underlying OS. To grant permission to read the value of only the `os.name` property, you would create a new `PropertyPermission` with the name "`os.name`". If you wanted to grant permission to read *all* OS properties, you would use the wild-card name "`os.*`".

For those instances where you do not need to use the actions property of a `Permission`, we recommend extending `java.security.BasicPermission`.

### 7.3.2 ActionsPermission

Permissions often specify actions that the grantee may perform. For example, a `java.io.FilePermission` specifies not only a file (the `Permission`'s name) for a `Permission`, but also what actions may be performed on that file, such as permission to read, write to, and delete the file. The SDK does not provide a base `java.security.Permission` class that uses actions. But, because actions are typically represented by a comma-separated list of action names, it's easy to provide a base class for action-based permissions. In this section, we provide such an implementation, `chp07.ActionsPermission`.

Our class must implement the four abstract methods on `java.security.Permission`, `equals()`, `getActions()`, `hashCode()`, and `implies()`. The implementation of equals() and hashCode() is straight forward, and follows the same code-pattern as the other base classes in this chapter: `equals()` tests for class equality, and then tests for member equality; `hashCode()`bases the hash value of the class and member values.

The `getActions()` implementation returns the canonical representation of the actions, which contains each of the actions passed into the constructor, in natural order, separated by a comma. The `implies()` method returns true if both the class type and name of the passed in permission is the same, and if the actions are a sub-set of the actions at hand. For example, the code below will output "`Implies? true`":

```
TestActionsPermission superSet = new TestActionsPermission("name",
"create, read");
TestActionsPermission subSet = new TestActionsPermission("name",
"create");

System.out.println("Implies? "+superSet.implies(subSet));
```

The code for `chp07.ActionsPermission` is below:

```
package chp07;

import java.security.Permission;
import java.util.Collections;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

public abstract class ActionsPermission
    extends Permission {

  private Set actionSet;
  private String actions;

  public ActionsPermission(String name, String actions) {
    super(name);
    if (name == null) {
      throw new NullPointerException(
          "permission name may not be null.");
    }

    actionSet = splitActions(actions);
    this.actions = canonizeActions(actionSet);
  }

  public String getActions() {
    return actions;
  }

  public boolean hasAction(String action) {
    return actionSet.contains(action);
  }
```

```
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }

  if (obj.getClass() != ActionsPermission.class) {
    return false;
  }

  ActionsPermission other = (ActionsPermission) obj;

  return getName().equals(other.getName())
      && getActions().equals(other.getActions());

}

public int hashCode() {
  return getClass().getName().hashCode() * 19
      + getName().hashCode() * 19 + getActions().hashCode() * 19;
}

public boolean implies(Permission permission) {
  // Test: this implies passed in permission?
  // i.e., passed in permission is a sub-set of this.
  if (equals(permission)) {
    return true;
  }

  if (getClass() != permission.getClass()) {
    return false;
  }

  ActionsPermission other = (ActionsPermission) permission;
  if (!getName().equals(other.getName())) {
    return false;
  }

  if (!actionSet.containsAll(other.actionSet)) {
    return false;
  }

  return true;
}

private Set splitActions(String actions) {
  Set actionSet = Collections.EMPTY_SET;
  if (actions != null && actions.trim().length() > 0) {
    actionSet = new TreeSet();
    String[] split = actions.split(",");
    for (int i = 0; i < split.length; i++) {
      String action = split[i];
      actionSet.add(action.trim());
      }
    }
    return actionSet;
  }

  private String canonizeActions(Set actions) {
    if (actions == null || actions.isEmpty()) {
      return "";
    }

    StringBuffer buf = new StringBuffer();
    for (Iterator itr = actions.iterator(); itr.hasNext();) {
      String action = (String) itr.next();
      buf.append(action);
      if (itr.hasNext()) {
        buf.append(",");
      }
    }

    return buf.toString();
  }

}
```

The next chapter contains an example of using `ActionsPermission`.

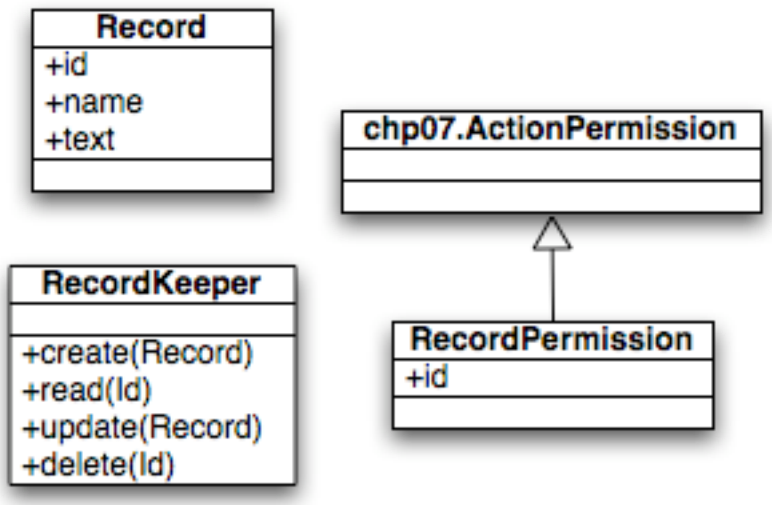
## *Summary*

As we've seen in previous chapters, JAAS is composed of several interfaces and base classes that you'll find yourself implementing and extending again and again. Unless you have a set of base cases to take care of the repetitive, but needed basic code–such as `toString()`, `equals()`, and `hashCode()`–you'll end up implementing the same functionality several times over. The base classes provided in this chapter for `Principals`, credentials, `CallbackHandlers`, and `Permissions` provided this set of base classes, allowing you to focus on the business logic of your application's security instead of the tedious plumbing.

# 8 JAAS for Data Access Control

This chapter is an overview of using JAAS to protect access to specific instances or pieces of data. For example, in a system where you have employee records, you might want to restrict access to those files based on who is trying to view or modify those records. This is called "data access control." JAAS is very good at, and commonly used to providing permissions for broad, class-level actions in systems, for example, controlling which users may change the system-wide `java.security.Policy` in effect. However, it's less common to see JAAS used to protect access to specific instances of classes. This chapter goes over using JAAS in this respect, using a custom `java.security.Permission` class to protect access to class instances.

## 8.2 The Record Domain



To demonstrate data access control, we'll use a simple domain of generic records. Each `Record` has a unique ID, a name, and text content. The records are immutable, and are maintained by the `RecordKeeper`. The `RecordKeeper` is a simplistic service that that manages the persistence and data access control for records. Before creating, reading, updating, or deleting any Record, the `RecordKeeperService` does a security check to verify that the current security context has been granted the appropriate `RecordPermision`, a custom `java.security.Permission` class used to protect `Records`.

The RecordPermission extends the ActionsPermission from chapter 7, providing an implementation of implies() that uses RecordPermission's actions. `RecordPermission`'s actions are used to specify which CRUD operation (create, read, update, or delete) the permission is granting. For example, to create an instance of `RecordPermission` that granted the right to read and update a Record, you would use the following code:

```
RecordPermission perm = new RecordPermission(id, "read, update");
```

## *8.1.2 Record*

The record class is a simple, immutable data object:

```
package chp08;

import util.id.Id;

public class Record
{

  public Id id;
  public String name;
  public String text;

  public Record(Id id, String name, String text)
  {
   this.id = id;
   this.name = name;
   this.text = text;
  }

  public Id getId()
  {
    return id;
  }

  public String getName()
  {
    return name;
  }

  public String getText()
  {
    return text;
  }
}
```

## *8.1.3 RecordKeeper*

`RecordKeeper` is a simplistic service that manages the life of `Record` instances. To keep this example simple, `RecordKeeper` stores Records in memory instead of in a more durable persistence store, like a database. `RecordKeeper` stores `Record` instances in a `java.util.HashMap`, where the key is the Record's ID and the value is the Record instance itself. Before performing any of the four data modification actions, `RecordKeeper` creates a `RecordPermission` to verify that the current security context has been granted the appropriate `RecordPermission`.

The code for `RecordKeeper` is below:

```
package chp08;

import java.security.AccessController;
import java.util.HashMap;
import java.util.Map;

import util.id.Id;

public final class RecordKeeper {

  private Map records = new HashMap();

  public void create(Record record) {
    RecordPermission perm = new RecordPermission(record.getId(),
        "create");
    AccessController.checkPermission(perm);
    records.put(record.getId(), record);
  }

  public Record read(Id recordId) {
    RecordPermission perm = new RecordPermission(recordId, "read");
    AccessController.checkPermission(perm);
    return (Record) records.get(recordId);
  }

  public void update(Record record) {
    RecordPermission perm = new RecordPermission(record.getId(),
        "update");
    AccessController.checkPermission(perm);
    records.put(record.getId(), record);
  }

  public void delete(Id recordId) {
    RecordPermission perm = new RecordPermission(recordId, "delete");
    AccessController.checkPermission(perm);
    records.remove(recordId);
  }
}
```

### *8.1.4 RecordPermission*

The custom permission used to protected `Records` uses the permission's name to specify the `Record`'s unique `Id`, and the actions attribute to specify if permission is granted to create, read, update, or delete the `Record` in question. Because `RecordPermission` extends `ActionsPermission`, all we need to implement in the code is a constructor that takes the Id of the `Record` being protected and the actions granted for that `Record`:

```
package chp08;

import util.id.Id;
import chp07.ActionsPermission;

public class RecordPermission
    extends ActionsPermission {

  public RecordPermission(String recordId, String actions) {
    super(recordId, actions);
  }

  public RecordPermission(Id recordId, String actions) {
    this(recordId.getId(), actions);
  }
}
```

## 8.2 JAAS Code

We'll use the authentication code developed in chapter 4 along with the database-backed authorization code developed in chapter 6. Recall that the authorization code stores `Permission` grants in a table with columns for the `Permission` type, a unique ID for the `Permission`, the `Permission`'s name, and the `Permission`'s actions.

## 8.3 The Example Application

Once again, we'll use a script-class with a main method to demonstrate this chapter's code in action. The "application" first configures JAAS to use our custom authentication and authorization code, then attempts to use the `RecordKeeper` without any `RecordPermission` being granted, grants the needed `RecordPermission`, and then uses `RecordKeeper`.

The code is below:

```
package chp08;

import java.security.Policy;
import java.security.PrivilegedAction;

import javax.security.auth.Subject;

import util.id.Id;
import chp04.UserGroupPrincipal;
import chp06.AuthHelper;
import chp06.DbPolicy;
import chp06.PermissionService;

public class Main {
```

```
public static void main(String[] args) throws Exception {
  AuthHelper authHelper = new AuthHelper();

  try {
    authHelper.createTestUser("testuser", "password");
    UserGroupPrincipal p = authHelper.getUserGrp();
    authHelper.loginTestUser();
    Subject subject = authHelper.getSubject();

    Policy.setPolicy(new DbPolicy());
    boolean granted = true;
    try {
      Subject.doAsPrivileged(subject, new PrivilegedAction() {

        public Object run() {
          Record r = new Record(Id.create("id1"), "record1",
              "this is some content.");
          RecordKeeper rk = new RecordKeeper();
          rk.create(r);
          return null;
        }
      }, null);
    } catch (SecurityException e) {
      granted = false;
    }

    System.out.println("Permission granted? " + granted);

    RecordPermission perm = new RecordPermission(
        Id.create("id1"), "create,read");
    PermissionService.addPermission(p.getId(), Id
        .create("permId1"), perm);
    System.out.println("Added grant for RecordPermission.");
    granted = true;
    try {
      Subject.doAsPrivileged(subject, new PrivilegedAction() {

        public Object run() {
          Record r = new Record(Id.create("id1"), "record1",
              "this is some content.");
          RecordKeeper rk = new RecordKeeper();
          rk.create(r);
          return null;
        }
```

```
          }, null);
        } catch (SecurityException e) {
          granted = false;
        }
        System.out.println("Permission granted? " + granted);
      } finally {
        authHelper.cleanUp();
        PermissionService.removePermission(Id.create("permId1"));
      }
    }

  }
```

To run the above, change to the root directory of this book's project, and type `ant run-chp08`.The output of will include the following:

```
Permission granted? false
Added grant for RecordPermission.
Permission granted? true
```

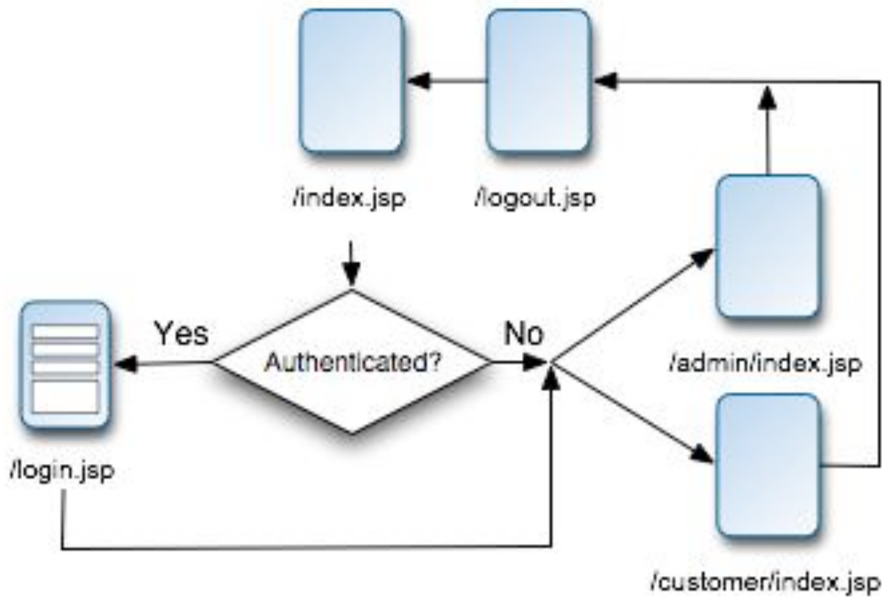
## *Summary*

This chapter provided a concise example of using JAAS for data access control. When you're restricting access to specific instance of object or data, not just general, system-wide actions, you're doing data access control. For our example, we created a small data object that represented a `Record`, and a service layer, `RecordKeeper`, that performed persistence and lookup of and for that data object. Also, we created a custom `Permision`, `RecordPermision`, that was used by `RecordKeeper` to restrict the actions of creating, reading, updating, and deleting specific `Records`. As the demonstration at the end of this chapter showed, this powerful, yet simple model allowed us to easily control access to each individual `Record`.

# *9 JAAS in Web Applications*

Though the Servlet spec doesn't officially integrate with JAAS, by convention, most Servlet containers provide several JAAS-related functions: restricting access to pages in a web application, providing a framework to authenticate users, and methods to access authentication information. Pages restrictions are specified by URL patterns and a list of "role" names that the requesting user must have to access the URLs. How these role names map to `Principals` is not specified, but in Tomcat, the role names are simply the `String` names of `Principals`. The framework for authenticating users can be used to create login screens that gather a user's username and password, and then associate the authenticated user with the session. The methods for accessing authentication information allow you to programmatically verify which `Principals` a user is in, retrieve their Servlet-centric `Principals`, and access other security-related state.

## *9.1 The Web Application*



This chapter uses a simple web application, diagramed above, with a handful of pages to demonstrate each of the above integration points between JAAS and Servlets. The web application provides a home page with links to an admin page, a customer page, and support pages to log in users, log out users, and an error page. As their names suggest, a user must be authenticated as an admin to access the admin page and a customer to access the customer page. Also, this chapter discusses a simple custom tag library that displays it's body content based on an authenticated user's `Principal` set.

## *9.2 Configuring JAAS with Servlets*

To enable JAAS in a web application, three things must be configured. First, the web container must be configured to create a "realm" that will be used to authenticate users. The Servlet spec does not specify how this configuration is done, so it's different for each web container. Once a realm is setup, the `web.xml` file must be modified to enable the authentication framework and to include mappings of URL patterns to the `Principal` names required to access those URLs. Finally, JAAS must itself be configured to specify the `LoginModule` implementations to use when authenticating a user.

### *9.2.1 Configuring Realms*

A realm has one responsibility: authenticate a user based on a username and password, adding "roles" to that user if authentication was successful. The Servlet spec doesn't specify how this responsibility is implemented, or very many other semantics of realms except that a realm must be able to represent roles with `String` names. Because of this looseness, each web container implementation is able to provide many different realm implementations: simple flat-file based realms, LDAP or other directory-based realms, OS authentication realms, and many other methods. Practically ever web container also provides a way to use JAAS as a realm. In the instances when JAAS is used as a Servlet realm, the web container gathers a user's username and password credentials, and delegates authenticating the user to the JAAS authentication framework, using a `LoginContext` and `LoginModule` implementations.

In this chapter, we use Tomcat 5.0.28 as our web container. Tomcat is the reference implementation for the Servlet 2.4 specification, and it provides a simple way to use JAAS realms. Realms are configured in Tomcat by modifying either the system-wide `server.xml`, or the web application's uniquely named `server.xml`. In our example, we modify the second to keep our application as self-contained as possible.

#### *Modifying server.xml*

Web application `server.xml` files are stored in `<tomcat dir>/conf/Catalina/localhost/` and follow the convention of being named the same as their corresponding web application. Our web application is named `jaas-book-chp09`, so the `server.xml` file we're interested in is found at `<tomcat dir>/conf/Catalina/localhost/jaas-book-chp09.xml`. The content of the file is below:

```
<?xml version="1.0"?>
<Context path="/jaas-book-chp09" docBase="~/tomcat/webapps/jaas-book-
chp09"
         debug="0" reloadable="true">
  <Realm className="org.apache.catalina.realm.JAASRealm" #1
         appName="chp09" #2
         userClassNames="chp09.UserPrincipal" |#3
         roleClassNames="chp04.UserGroupPrincipal" |#3
         useContextClassLoader="false"/> #4
</Context>
```

(annotation) <#1: The realm tag specifies that Tomcat's **`JAASRealm`** will be used to authenticate users.
(annotation) <#2: appName is used to specify which **`LoginModule`** group will be used to authenticate users. "chp09" is the application name that will be passed into the **`LoginContext`** constructor. So, we'll have to ensure that our **`javax.security.login.Configuration`** can return an **`AppConfigurationEntry`** array for that application name.>
(annotation) <#3: The **`userClassNames`** and **`roleClassNames`** attributes specify which **`Principal`** implementations will be used to represent the user **`Principal`** and the role **`Principals`**. Once a user has been authenticated, creating a **`Subject`** with **`Principals`**, the user **`Principal`** is used when the web container looks up the **`Subject`**'s user, for example, when looking up the value for **`HttpServletRequest`**'s **`getRemoteUser()`** or **`getUserPrincipal()`**. The role **`Principals`** are used to lookup the **`Subject`**'s roles, for example, when resolving if a user is in a role for **`HttpServletRequest`**'s **`isUserInRole.>`**
(annotation) <#4: setting this attribute to **`false`** tells Tomcat to use the web application's class loader instead of the server class loader. The **`LoginModule`** implementation we'll be using (**`DbLoginModule`** and **`TomcatLoginModule`**) will be stored in the web application's lib directory, meaning that the server level class loader will not be able to find it.>

With `chp09-server.xml` in place, Tomcat will create an authentication realm that will be used once we configure `web.xml` to enable security.

## *9.2.2 Configuring web.xml*

As with practically ever other feature in Servlets, enabling authentication and authorization is done by modifying the `web.xml` file. Three tags are used to enable authentication, specify URL access restrictions, and define the available user roles, or `Principals`.

### *Enabling Authentication*

Web containers may provide five types of authentication schemes: BASIC and DIGEST, which use the built in username and password dialog box for HTTP; FORM, which uses custom JSP pages with standard form action and element names; CLIENT-CERT, which uses digital certificates; and any proprietary mechanisms that the web container provides. In this book, we'll only cover the use of the FORM method because it covers the widest range of cases and allows for a fair amount of customization.

In our example web application, configuring authorization in `web.xml` is done with the following element:

```
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>Chp09 Realm</realm-name> #1
    <form-login-config> #2
     <form-login-page>/login.jsp</form-login-page>
     <form-error-page>/login-error.jsp</form-error-page>
    </form-login-config>
  </login-config>
```

(annotation <#1: the realm name is used purely for display purposes, mostly for web application development tools.>

**(annotation) <#2: this element and its sub-elements specify the location of the login page to use and JSP page to use when authentication errors occur. The error page is used for invalid login attempts and when errors occur logging in. A different page, covered below, is used when a user attempts to access a URL they're not authorized to view.>**

## *Locking Down URLs with security-constraints*

To specify access control for parts of your web application, you use any number of `security-constraint` elements. The `security-constraint` element specifies one or more URL patterns and the `Principals`, represented by role names, a user is required to have to access those URLs. When an unauthenticated user attempts to request one of the protected URL, the web container redirects the request to the login page specified in the `login-config` element in `web.xml`.

A URL pattern can be an exact match, like `/admin/userlist.jsp`, or a pattern, like `/admin/*`. The first pattern specifies a single page, while the second specifies any URLs that begin with `/admin`. The patterns are all relative to the web application context.

The role names specified may either be the `String` name of a `Principal`, or the special role name *, which is shorthand for any role. When a user requests a URL specified by the `security-constraints` URL patterns, the users `Subject` must have one of the roles specified, or access is denied.

The security-constraint elements used in our example `web.xml` are below:

```
<security-constraint>
    <web-resource-collection>
      <web-resource-name>Admin Page (Chp09)</web-resource-name>
      <url-pattern>/admin/*</url-pattern> #1
      </web-resource-collection>
    <auth-constraint>
       <role-name>admin</role-name> #2
    </auth-constraint>
  </security-constraint>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Customer Page (Chp09)</web-resource-name>
      <url-pattern>/customer/*</url-pattern> #1
      </web-resource-collection>
    <auth-constraint>
       <role-name>customer</role-name> #2
    </auth-constraint>
  </security-constraint>
```

**(annotation) <#1: these tags specify the URLs that have access restrictions. The URL patterns used in this example cover all URLs that begin with `/admin` and all URLs that begin with `/customer`. Each URL is relative to the web application context.>**
**(annotation) <#2: the `Principals` required to access restricted URLs are specified in the `auth-constraint` element by `role-name` elements. Any number of `role-name` elements may be specified. Each `role-name` element specifies one `Principal`, by name, that may access the restricted URL. If none are specified, then no users may access the restricted URLs, preventing all access to the URLs.>**

## *Specifying the roles Used*

Finally, you must specify all of the role names that are referenced in `web.xml` with one `security-role` element per role. Specifying each role that's used seems tedious, but it allows tools to get a list of roles used and provides crude referential integrity[1].

The security-role elements for our example web application are below:

```
  <security-role>
    <description>
      Role required to see admin pages.
    </description>
    <role-name>admin</role-name>
  </security-role>

  <security-role>
    <description>
      Role required to see customer pages.
    </description>
    <role-name>customer</role-name>
  </security-role>
```

## *Other Settings and Entire web.xml Listing*

The entire `web.xml` listing is below, with code notations for the remainder of the settings:

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
  <display-name>jaas-book</display-name>
  <description>JAAS Book, Chapter 9</description>

  <servlet>
    <servlet-name>InitServlet</servlet-name> #1
    <servlet-class>chp09.StartupServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
  </servlet>

   <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>

  <error-page> #2
    <error-code>403</error-code>
    <location>/access-denied.jsp</location>
  </error-page>
```

[1] Tomcat allows you to skip specifying the security-role element, but logs an error if you omit them. Other web containers may not be so forgiving.

```
  <taglib> #3
    <taglib-uri>auth-tags</taglib-uri>
    <taglib-location>/WEB-INF/auth-tags.tld</taglib-location>
  </taglib>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Admin Page (Chp09)</web-resource-name>
      <url-pattern>/admin/*</url-pattern>
      </web-resource-collection>
    <auth-constraint>
       <role-name>admin</role-name>
    </auth-constraint>
  </security-constraint>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Customer Page (Chp09)</web-resource-name>
      <url-pattern>/customer/*</url-pattern>
      </web-resource-collection>
    <auth-constraint>
       <role-name>customer</role-name>
    </auth-constraint>
  </security-constraint>

  <login-config>
    <auth-method>FORM</auth-method>
    <realm-name>Chp09 Realm</realm-name>
    <form-login-config>
     <form-login-page>/login.jsp</form-login-page>
     <form-error-page>/login-error.jsp</form-error-page>
    </form-login-config>
  </login-config>

  <security-role>
    <description>
      Role required to see admin pages.
    </description>
    <role-name>admin</role-name>
  </security-role>

  <security-role>
    <description>
      Role required to see customer pages.
    </description>
    <role-name>customer</role-name>
  </security-role>

</web-app>
```

(annotation) <#1 [InitServlet]: the `InitServlet` is used to configure `DbConfiguration` and configure logging. See section XXX below for further discusion of the InitServlet.>

**(annotation) <#2 [error-page]: this `error-page` element specified the page to use when a user attempts to access a URL they are not authorized to view. If this page is not specified, a generic error page is used instead. We'll see this page in action below when we walk through the web application's pages.>**
**(annotation) <#3 [taglib]: this specifies the custom tag library that contains the role display tag, seen later in this chapter.>**

## *9.2.3 The JSP Pages*

Our `web.xml` file references several JSP pages: `login.jsp`, `login-error.jsp`, and `access-denied.jsp`. The `login.jsp` page is used to gather username and password credentials, and the `login-error.jsp` when a user fails authentication or an error occurs authenticating a user. The last page, `access-denied.jsp`, is used when an authenticated user attempts to access a page that requires a `Principal` the user doesn't have.

In addition to these 3 pages, 3 other pages are included in the example web application: `logout.jsp` which invalidates the user's session, thus logging out the user; the top level `index.jsp` which has links to the Admin and Customer page, and a link to `logout.jsp`; and an `index.jsp` pages for the `admin` and `customer` sub-directories.

Of these JSP pages, the only noteworthy pages are `login.jsp` and `logout.jsp`. At the end of this chapter, in the `RoleTag` section, we'll go over the top-level `index.jsp`. The other JSPs are available in the accompanying source, and we won't go over them here.

### *login.jsp*

The `login.jsp` page contains the custom login form used in our web application. The Servlet spec requires that the action for the login form be `j_security_check`. Also, the form input field for the username must have the name `j_username`, and the password input field must have the name `j_password`. Requiring these names makes implementing Servlet security a little easier for web container vendors, and isn't too much of an inconvenience for developers.

The source for `login.jsp` is listed below:

```
<html>
<head><title>Chapter 09 Login</title></head>
<body>

<form method="POST" action="j_security_check">
<p>Username: <input type="text" name="j_username"/></p>
<p>Password: <input type="password" name="j_password"/></p>
<input type="submit" value="Login"/>
</form>

</body>
</html>
```

### *logout.jsp*

`logout.jsp` is interesting because it contains code that logs out the currently authenticated user. The convention for logging out users is to invalidate the user's session. The `logout.jsp` does this with a small inline code fragment; this code could be done in a Servlet, Struts Action, or other non-JSP code just as easily.

The code listing is below:

```
<html>
<head><title>Chp09 Logout</title></head>
<body>

<%
try {
  session.invalidate();
}
catch(IllegalStateException e) {
  // we don't care
}
%>

<p>You've been logged out.</p>
<p><a href="<%= request.getContextPath() %>/index.jsp">Home</a></p>

</body>
</html>
```

### *9.2.4 Configuring JAAS*

JAAS authorization services must be enabled for Tomcat's JAASRealm to work. This is done through the standard methods of using either VM arguments, modifying the VM's `security.properties` file, or programmatically setting the `javax.security.auth.Configuration` to use. In our application, we use `DbConfiguration`, from chapter 4, as our `Configuration` implementation. To make the web application more self-contained, we programmatically set the `Configuration` by calling `DbConfiguration`'s `init()` method in a startup Servlet.

A startup Servlet is a Servlet that the web container loads immediately after loading the web application. By overriding the `init()` method, you can programmatically configure your web application. We use this pattern to configure `DbConfiguration` and to configure JDK logging in our web application. The Servlet is specified and configured in `web.xml`.

The code for the startup Servlet is below:

```
package chp09;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;

import util.LoggerInit;

import chp04.DbConfiguration;

public class StartupServlet
    extends HttpServlet {

  public void init(ServletConfig config) throws ServletException {
    DbConfiguration.init();
    LoggerInit.init();
  }
}
```

When our example web application loads, before serving any requests, the above code is executed, configuring `DbConfiguration` to be used by JAAS.

## *TomcatLoginModule*

With JAAS configured, we now need to specify a `LoginModule` to use when authenticating users. We can re-use the functionality of chapter 4's `DbLoginModule`, allowing us to specify and manage users and their role-`Principals` in a database. However, we need to add an additional `Principal` that represents the authenticated `Subject`'s user. Effectively, this Principal simply wraps the username entered in the login page[2]. As mentioned above, this `Principal` is returned from `HttpServletRequest`'s `getUserPrincipal()` method.

To add the user `Principal`, we create a new `LoginModule` implementation, `TomcatLoginModule`, which extends `DbLoginModule` and overrides the `commit()` method. The result is that the `Subject` is given all of the `Principals` assigned to it in the database in addition to the special user `Principal` needed by Tomcat.

```
package chp09;

import javax.security.auth.login.LoginException;

import chp04.DbLoginModule;

public class TomcatLoginModule
    extends DbLoginModule {

  public boolean commit() throws LoginException {
    if (super.commit()) {
      UserPrincipal userP = new UserPrincipal(getUsername()); #1
      getSubject().getPrincipals().add(userP);
      getPrincipalsAdded().add(userP);
      return true;
    } else {
      return false;
    }
  }
}
```

**(annotation) <#1: `chp09.UserPrincipal` is the `Principal` we specified when configuring the realm for Tomcat. The user `Principal` simply wraps the username provided by the user.>**

## *Database Setup*

[2] If a different mapping makes more sense for your web application, the Servlet spec does not require that the user Principal's name is the same as the username. In our example, and most web applications, wrapping the username is sufficient.

To support `DbConfiguration` and `TomcatLoginModule`, we setup the same database we used in chapter 4, seeding it with rows for the customer and admin users, and then adding the corresponding `RolePrincipal` to each:

```
INSERT INTO app_configuration VALUES
('chp09', 'chp09.TomcatLoginModule', 'REQUIRED');

INSERT INTO db_user VALUES
('admin-user-id', 'admin', 'secret');

INSERT INTO db_user VALUES
('customer-user-id', 'customer', 'secret');

INSERT INTO principal VALUES
('admin-principal-id',
'admin',
'chp09.RolePrincipal');

INSERT INTO principal_db_user VALUES
('admin-user-id','admin-principal-id');

INSERT INTO principal VALUES
('customer-principal-id',
'customer',
'chp09.RolePrincipal');

INSERT INTO principal_db_user VALUES
('customer-user-id','customer-principal-id')
```

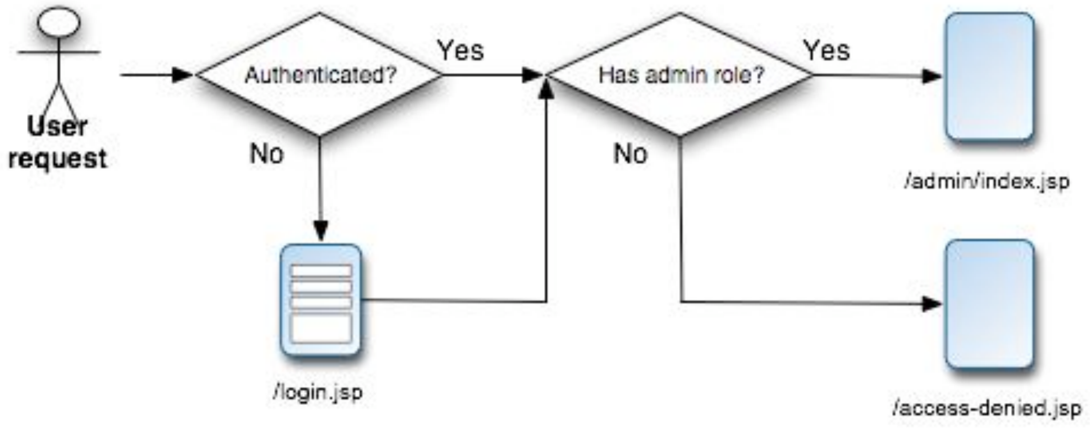
## *9.3 The Web Application*

Once the above configurations and other setup are done, we're ready to use our example web application, demonstrating how JAAS can be used with Servlets to restrict access to sections of the web application. This section walks through the pages of the example web application, demonstrating how the web container uses the configuration and code in the preceding sections.

### *9.3.1 URL Access Control*

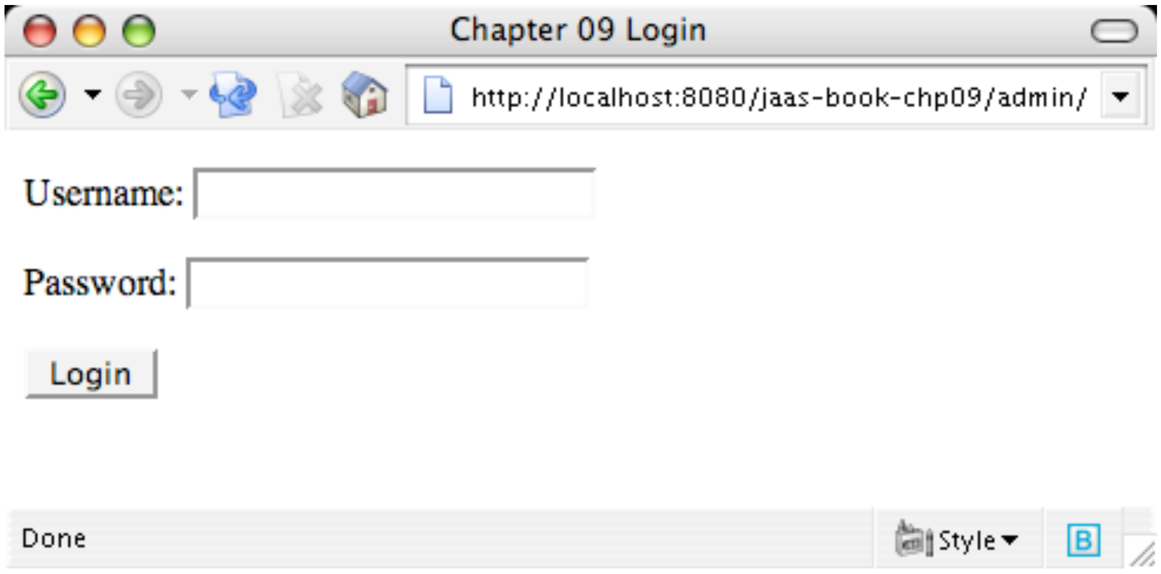
Each time a restricted URL is requested, the web container first ensures that a user is logged in, redirecting the request to the `login.jsp` if there is no user associated with the current session. Once a user has been logged in, the web container will then see if the user belongs to any of the role-`Principals` that are allowed to access the restricted URL, specified by the `security-constraint` element in `web.xml`. If the user does belong to one of those roles, they're allowed to access the URL. Otherwise, if the user does not belong to one of those roles, they're forwarded to the 403 error page, specified by the `error-page` element in `web.xml`.

The diagram below illustrates this flow:

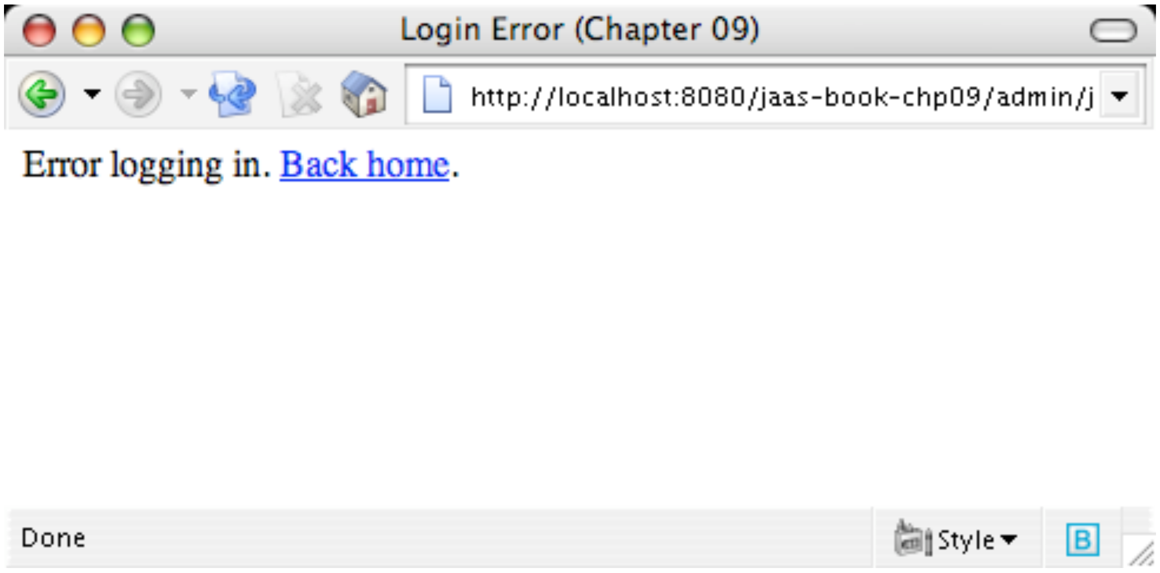


## *9.3.2 Example: Accessing /admin/index.jsp*

Let's suppose that we have a user who wants to access the admin page, `/admin/index.jsp` in our web application. The user hasn't been authenticated yet, so when he first attempts to access the page, he'll be redirected to `login.jsp`:

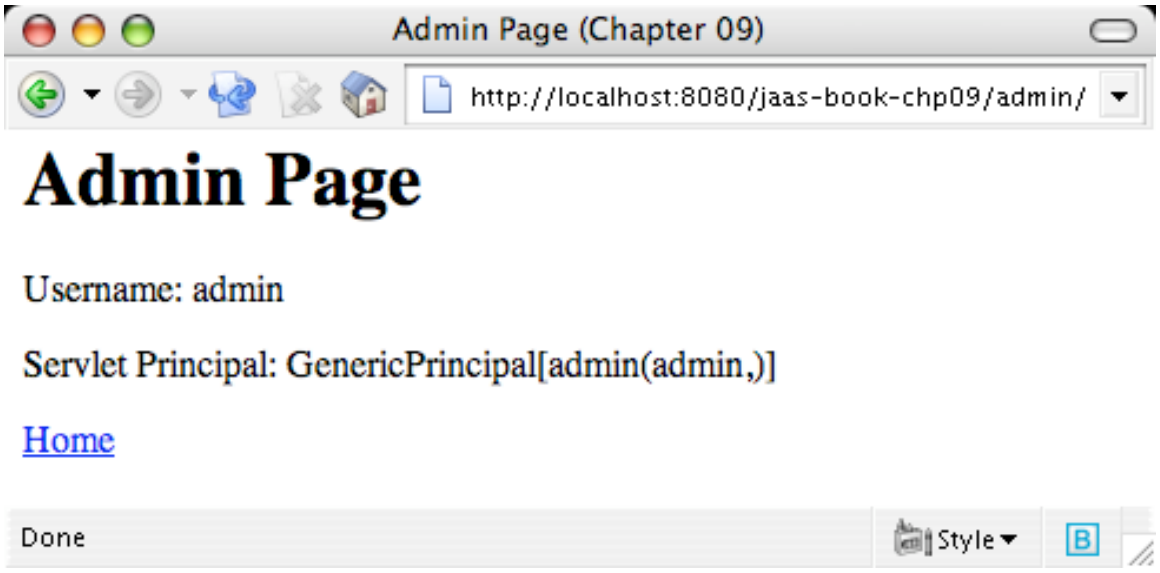


The user types in the correct username, admin, but uses the incorrect password. When `TomcatLoginModule` is invoked to authenticate the user, it throws a `LoginException` from it's `login()` method, causing the web container to forward to the `login-error.jsp` page:



Error logging in. Back home.

The user realizes their mistake, goes back to the login page, and enters the correct password. With the correct password, `TomcatLoginModule`'s `login()` method returns `true`, causing `commit()` to be called, successfully authenticating the user and adding the required admin `RolePrincipal` to the user's `Subject`. Because the user has now been authenticated and has the required admin `RolePrincipal`, the web container forwards them to access the originally requested page, `/admin/index.jsp`:



Admin Page

Username: admin

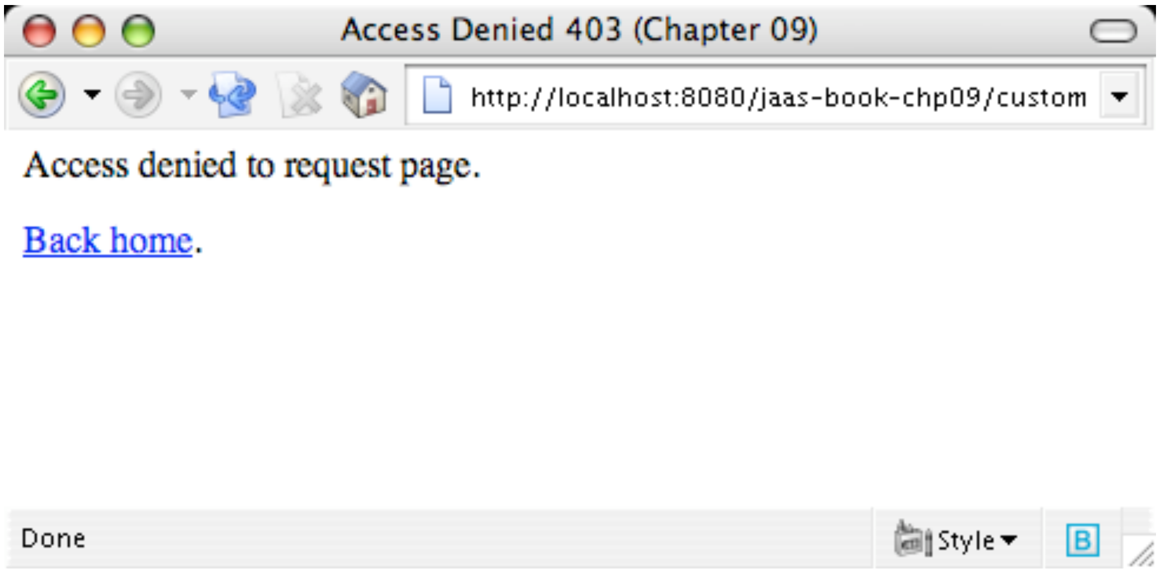
Servlet Principal: GenericPrincipal[admin(admin,)]

Home

The page displays the username and the `toString()` value of the `Principal` returned from `HttpRequest`'s method `getUserPrincipal()`.

### *9.3.3 Example: Accessing /customer/index.jsp*

Next, the user attempts to access the customer page by going to the URL `/customer/index.jsp`. The security-constraint for this page requires the user to have the customer `RolePrincipal` which the admin user does not have. So, the web container redirects the user to 403 error page as specified by the error-page element in `web.xml`, `access-denied.jsp`:


Access Denied 403 (Chapter 09)
http://localhost:8080/jaas-book-chp09/custom
Done
Style

Access denied to request page.

Back home.

## *9.4 RolesTag*

To demonstrate programmatically some of a Servlet's JAAS-related methods, we'll develop a custom tag library that displays the body of the tag only if the authenticated user has one of the `Principals` the tag specifies. This is a very common scenario. For example, we may have a section of the page that we only want users with the admin `RolePrincipal` to see.

The tag's only attribute, `roles`, specifies a comma-separated list of `Principals` that the user must have to see the body of the tag. The body of the tag will be displayed if the authenticated user has at least one of the `Principals` specified by the `roles` attribute.

### *9.4.2 A Pure Scriptlet Implementation*

To appreciate the utility of having a tag to perform role checks, we'll first look at how we'd check for a user's roles purely with JSP scriptlets:

```
<%@ taglib uri="auth-tags" prefix="auth" %>
<html>
<head><title>Index</title></head>
<body>

<a href="admin">Admin Page</a> |
<a href="customer">Customer Page</a> |
<a href="logout.jsp">Logout</a>

<% if (request.isUserInRole("customer")) { %> #1
<p>Only the <b>customer</b> role sees this.</p>
<% } %>

<% if (request.isUserInRole("admin") ||
       request.isUserInRole("superadmin")) { %> #2
<p>Only the <b>admin</b> role sees this.</p>
<% } %>

<p>Principals: <%= request.getUserPrincipal() %>.</p>

</body>
</html>
```

**(annotation) <#1 HttpServletRequest provides the method isUserInRole which returns true if the currently logged in user has the passed in role.>**
**(annotation) <#2 isUserInRole only accepts one role at a time, so to check for multiple roles, you have to or together a call for each role.>**

While using a pure scriptlet approach doesn't require any extra code or configuration files (as the below taglib approach does), it suffers a key disadvantages: lack of abstraction. Instead of layer how our web application does security checks, we're directly coding that method into our JSP page. If we later decide to check for a user's role using a different way than using the isUserInRole() method, we'll have a lot of JSP pages to change. Aside from that, as with most scriptlet code, it just looks ugly.

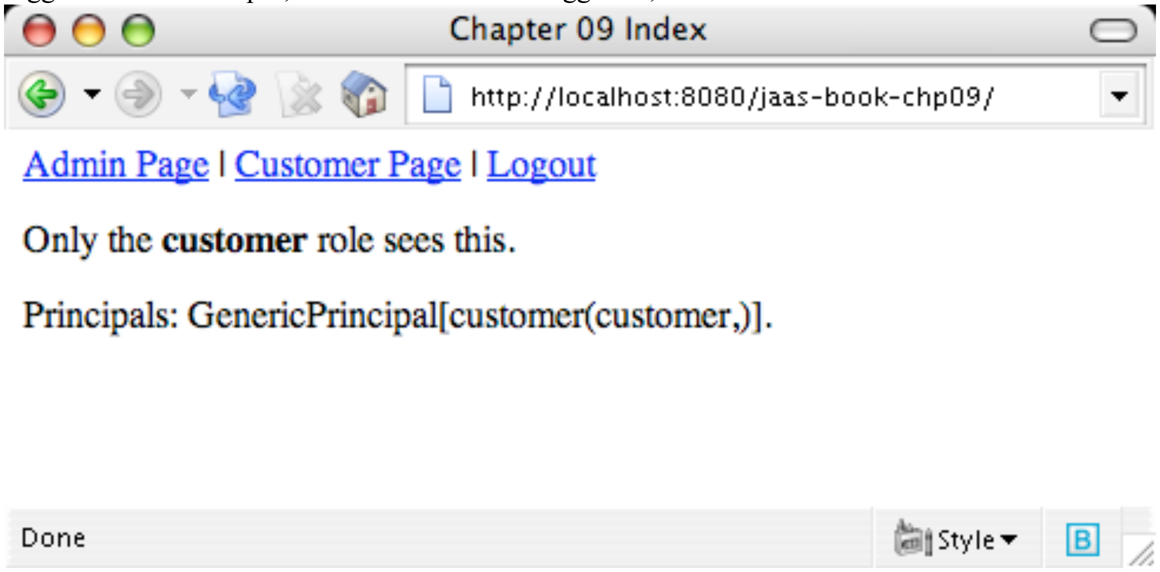
### *9.4.1 Using RolesTag*

The top-level `index.jsp` uses demonstrates the use of this tag:

```
<%@ taglib uri="auth-tags" prefix="auth" %>
<html>
<head><title>Chapter 09 Index</title></head>
<body>

<a href="admin">Admin Page</a> |
<a href="customer">Customer Page</a> |
<a href="logout.jsp">Logout</a>

<auth:roles roles="customer">
<p>Only the <b>customer</b> role sees this.</p>
</auth:roles>

<auth:roles roles="admin">
<p>Only the <b>admin</b> role sees this.</p>
</auth:roles>

<p>Principals: <%= request.getUserPrincipal() %>.</p>
```

```
</body>
</html>
```

The `RolesTag` is used twice in this page. The first instance creates a section of the JSP page that will only be displayed when a user with the customer `RolePrincipal` is logged in, while the second displays it's body content only when a user with the admin `RolePrincipal` is logged in. For example, when a customer is logged in, the JSP will be rendered as:



Admin Page | Customer Page | Logout

Only the **customer** role sees this.

Principals: GenericPrincipal[customer(customer,)].

## *9.4.2 RolesTag's TLD*

The following tag library descriptor configures the `RolesTag`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>auth</shortname>
  <uri>/WEB-INF/auth-tags.tld</uri>
  <tag>
    <name>roles</name>
    <tagclass>chp09.RolesTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <attribute>
      <name>roles</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>
</taglib>
```

We saw the tag library included with the `taglib` element in the complete listing of the web application's `web.xml` above.

### *RolesTag Code*

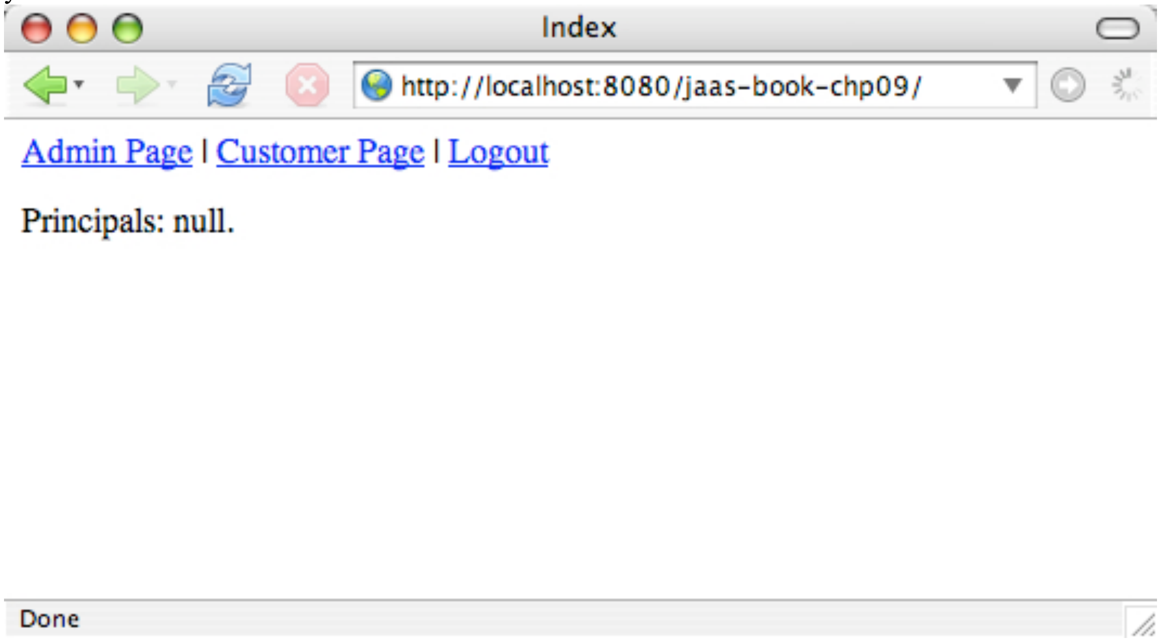
Once `RolesTag` verifies that a value for the roles attribute was specified, it splits the list of roles in an array of names. `RolesTag` then iterates over this array of names, passing each name to `isUserInRole()` on `HttpServletRequest`. If `isUserInRole()` returns `true`, `RolesTag` returns `INCLUDE_BODY`, causing the body of the tag to be displayed. Otherwise, if the user does not have one of the roles required, `SKIP_BODY` is returned causing the body of the tag to be omitted from the rendered JSP. If the user is not even authenticated, `isUserInRole()` will return `false` each time, causing the body of the tag to be omitted to un-authenticated users as well.

```
package chp09;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.jsp.tagext.TagSupport;

public class RolesTag
    extends TagSupport {

  public int doStartTag() {
    if (roles_ != null || roles_.length() != 0) {
      boolean userHasRole = false;
      HttpServletRequest request = (HttpServletRequest) pageContext
          .getRequest();
      String[] splitRoles = roles_.split(",");
      for (int i = 0; i < splitRoles.length; i++) {
        String role = splitRoles[i];
        if (request.isUserInRole(role.trim())) {
          return EVAL_BODY_INCLUDE;
        }

      }
    }
    return SKIP_BODY;
  }

  public String getRoles() {
    return roles_;
  }

  public void setRoles(String roles) {
    roles_ = roles;
  }

  private String roles_;
}
```

## *Running the Example Web Application*

To deploy the example web application for this chapter, change directories to the source code directory, and execute the command `ant deploy-chp09`. This will configure the database for you, compile the required code, and deploy the web application to Tomcat.

Once you start Tomcat, you'll be able to load the example web application by going to the URL `http://localhost:8080/jaas-book-chp09/` in your browser. The first page you'll see is below:



Admin Page | Customer Page | Logout

Principals: null.

From the index page, you can attempt to access both the Admin and Customer page. Once you click on either link, you'll be redirected to the login page which will prompt you for a username and password as seen in the screenshots in the previous sections. To login as an admin, use the credentials admin/secret; to login as a customer, use the credentials customer/secret.

## *Summary*

With a good understanding of JAAS under our belts, we were ready to start using JAAS in a web application. The first step using JAAS in a web application was modifying the application's `web.xml` file to enable authentication. Once authentication was enabled, we learned how to customize the different JSP pages used by the web container to log a user in and display error messages. With the configuration under our belts, we went over two simple ways to secure parts of any web applications: URL access restrictions and a custom tag library that conditionally displays it's JSP body according the roles the logged in `Subject` has.

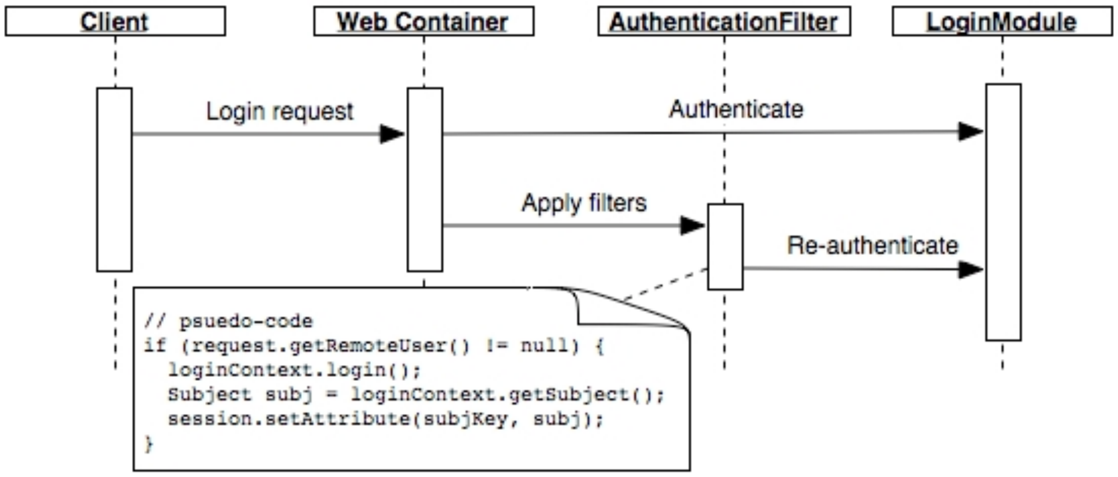
# *10 Extending JAAS Integration in Web Applications*

As the last chapter demonstrated, the Servlet specification provides an API for authenticating users, and testing the user's membership in roles that represent the user's `Principals`. Since JAAS is not tightly coupled to the Servlet spec, using the other features of JAAS, such as permission checking, are not as easily accomplished. To use all of JAAS, you need access to the authenticated `javax.security.auth.Subject` instance, allowing you to create `Subject`-based privileged blocks. This chapter demonstrates one way to get an authenticated `Subject`, and then goes over using that `Subject` to perform authorization checks.

## *10.1 The AuthenticationFilter*

If there is no other way for you to obtain the authenticated `Subject` from your web container, you can use a `ServletFilter` to create one yourself. In this strategy, the user is authenticated twice: once when the web container logs the user in where you *cannot* have access to the `Subject`, and again in a `ServletFilter` where you *can* access to the `Subject`. The second round of authentication is done automatically, without the user having to provide their credentials. The ability to do this relies on the way that `HttpServletRequest`'s `getRemoteUser()` works. The `getRemoteUser()` method returns the username of the authenticated user for the request. If there is no authenticated user, the method returns null. Thus, when `getRemoteUser()` returns a non-`null` value, you know that the container has already authenticated the user by prompting for the username and password. Knowing this, you can go through the process of authenticating the user without actually asking for their credentials. This step allows you to easily get access to an authenticated `Subject` instance by going through the process of authenticating a user without having to prompt the user for their username and password.

The diagram below illustrates the process:



```
// psuedo-code
if (request.getRemoteUser() != null) {
  loginContext.login();
  Subject subj = loginContext.getSubject();
  session.setAttribute(subjKey, subj);
}
```

When the client first logs in, the web container catches the special request (by looking for the action `j_security_check`) and uses your JAAS `LoginModule`(s) to authenticate the user. Once the user has been successfully authenticated, the web container will apply any `ServletFilters` configured. To re-authenticate the user, allowing us to acquire a reference to the authenticated `Subject`, we configure the `AuthenticationFilter` to be applied to all requests, meaning that the filter will be run after the web container has performed it's own authentication. As the pseudo-code in the diagram shows, each time the `AuthenticationFilter` runs, it checks if the user has been authenticated in the current request by calling `getRemoteUser()`. If the request is authenticated, the filter re-authenticates the user with it's own `LoginContext` instance, allowing the Filter to access the authenticated `Subject`. Once the `AuthenticationFilter` authenticates the `Subject`, it caches the `Subject` in the session so that other components in the web application can access the `Subject`.

The code for the `AuthenticationFilter` is below:

```
package chp10;

import java.io.IOException;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import chp04.UserService;
import chp07.BundleCallbackHandler;

public class AuthenticationFilter implements Filter {

  public void init(FilterConfig config) throws ServletException { #1
    appName_ = config.getInitParameter("app-name");
    subjectKey_ = config.getInitParameter("subject-key");
    if (subjectKey_ == null) {
      subjectKey_ = DEFAULT_SUBJECT_KEY;
    }
  }

  public void doFilter(ServletRequest request,
      ServletResponse response, FilterChain chain)
      throws IOException, ServletException {
    HttpServletRequest httpRequest = (HttpServletRequest) request;

    String remoteUser = httpRequest.getRemoteUser();
    if (remoteUser != null) { #2

      if (LOGGER.isLoggable(Level.FINE)) {
        Subject subj = (Subject) httpRequest.getSession()
            .getAttribute(subjectKey_);

        LOGGER.logp(Level.FINE, LOG_TOPIC, "doFilter()",
            "Subject found under key {0}:\n{1}", new Object[] {
                subjectKey_, subj });
      }

      String password = null;
      try {
        password = UserService.lookupPassword(remoteUser); #3
      } catch (SQLException e) {
        throw new ServletException(
            "Error retrieving credentials for " + remoteUser, e);
      }
      BundleCallbackHandler cb = new BundleCallbackHandler(
          remoteUser, password);
      try {
        LoginContext ctx = new LoginContext(appName_, cb);
        ctx.login();
        Subject subj = ctx.getSubject();

        httpRequest.getSession().setAttribute(subjectKey_, subj);

        LOGGER.info("Authenticated Subject " + subj
            + ". Under session key " + subjectKey_);

      } catch (LoginException e) {
        LOGGER
            .logp(
                Level.WARNING,
                LOG_TOPIC,
                "doFilter()",
                "LoginException thrown when validating user {0}.
Exception:\n{1}",
                new Object[] { remoteUser, e });
      }

    }

    chain.doFilter(request, response);
  }

  public void destroy() {
  }

  static private String LOG_TOPIC = AuthenticationFilter.class
      .getName();

  static private Logger LOGGER = Logger.getLogger(LOG_TOPIC);

  static private final String DEFAULT_SUBJECT_KEY = "subject";

  private String appName_;

  private String subjectKey_;

}
```

(annotation) <#1: [init()]:The `AuthenticationFilter` is configured with 2 filter parameters: the name of the JAAS `AppConfigurationEntry` group to use, and the session key to place the authenticated `Subject` under. A default value of "subject" is used for subject-key if that parameter isn't specified.>
(annotation) <#2: [Check for remoteUserName]: Authentication is done for every request, assuring that the changes to the Subject's Principal set take effect while the user is logged in.
(annotation) <#3: [Lookup password]: to simplify the example, the filter looks up a user's password so that we can reuse the same `TomcatLoginModule` as used in chapter 9. Alternatively, to avoid looking up the plaintext password, you could use a LoginModule that doesn't require a password to authenticate a user. The assumption with this type of LoginModule would be that the user was already authenticated.>

Sidebar: Obtaining the Subject in Different Application Servers.

Depending on the application server your application is running in, you may be able to use one of the below methods to get the authenticated Subject:

JBoss: the class `org.jboss.security.SecurityAssociation` provides the static method `getSubject()`.

WebSphere: the class `com.ibm.websphere.security.auth.WSSubject` provides the static method `getCallerSubject()`.

WebLogic: the class `weblogic.security.Security` provides the static method `getCurrentSubject()`.

End Sidebar

## *10.2 The DoAsPrivilegedFilter*

With a reference to the authenticated `Subject` available, our web application can use all of the features JAAS provides. To make checking permissions easier, we use another `ServletFilter` to wrap the entire request in a privileged block using the `Subject`'s static method `doAsPrivileged()`. Once this filter is applied, calls in code to `AccessController.checkPermission()` will use the authenticated `Subject` when checking for access.

The code for `DoAsPrivilegedFilter` is below:

```
package chp10;

import java.io.IOException;
import java.security.PrivilegedActionException;
import java.security.PrivilegedExceptionAction;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.security.auth.Subject;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;

public class DoAsPrivilegedFilter implements Filter {

  public void init(FilterConfig config) throws ServletException { #1
    subjectKey_ = config.getInitParameter("subject-key");
    if (subjectKey_ == null) {
      subjectKey_ = DEFAULT_SUBJECT_KEY;
    }
  }

  public void doFilter(final ServletRequest request,
      final ServletResponse response, final FilterChain chain)
      throws IOException, ServletException {
    HttpServletRequest httpRequest = (HttpServletRequest) request;
    Subject subj = (Subject) httpRequest.getSession().getAttribute(
        subjectKey_);
    if (subj == null) {
      LOGGER
          .logp(
              Level.FINE,
              LOG_TOPIC,
              "doFilter()",
              "No Subject found under key {0}, so creating "+
              "new Subject.",
              subjectKey_);
      subj = new Subject();
    }
    try {
      if (LOGGER.isLoggable(Level.FINE)) {
        LOGGER.logp(Level.FINE, LOG_TOPIC, "doFilter()",
            "Running doAsPrivileged block with Subject: {0}", subj);
      }

      Subject.doAsPrivileged(subj, new PrivilegedExceptionAction() { #2

        public Object run() throws Exception {
          chain.doFilter(request, response);
          return null;
        }

      }, null);
    } catch (PrivilegedActionException e) {
      LOGGER
          .logp(
              Level.SEVERE,
              LOG_TOPIC,
              "doFilter()",
              "Exception executing filter with Subject:\n"+
                                    {0}\nException: {1}",
              new Object[] { subj, e });
      throw new ServletException(e);
    }
  }

  public void destroy() {
  }

  static private String LOG_TOPIC = DoAsPrivilegedFilter.class
      .getName();
  static private Logger LOGGER = Logger.getLogger(LOG_TOPIC);
  static private final String DEFAULT_SUBJECT_KEY = "subject";
  private String subjectKey_;
}
```

(annotation) <#1 [init()]: **`DoAsPrivilegedFilter`** is configured with one optional init parameter, **`subject-key`**, which specifies session key to look for the authenticated Subject under. If this parameter is not specified, the default value of "subject" is used.>
(annotation) <#2 [Subject.doAsPrivileged() call]: **`DoAsPrivilegedFilter`** passes in a null **`AccessControllerContext`** as the third argument to **`doAsPrivileged`**, assuring that only authenticated **`Subject`** is used for authorization.>

### *10.2.1 Advantages & Limitations of the DoAsPrivilegedFilter*

Aesthetically, the `DoAsPrivilegedFilter` is appealing because it wraps an entire request in a secure block of code. The chain of code from the UI all the way down to the back-end is protected in this block. Also, instead of passing the `Subject` down the entire execution chain, you can rely on JAAS to provide the `Subject` as needed.

Unfortunately, wrapping your entire request in a privileged block can cause several challenging problems in an application that uses a lot of 3rd party libraries. Because the entire execution stack is executed in the privileged block, the `Subject` must be granted all the permissions needed by each piece of code, or each block of code must be wrapped in it's own privileged block and granted those permissions. With the amount of third party libraries in most Java programming, assuring either of these two states can be incredibly time consuming and tedious. Most Java code is not written with such JAAS-centric concerns in mind, let alone documented with the permissions needed for normal execution. The result is that if you want to wrap an entire request in a privileged block, you'll have several days, if not weeks, of tedious trial and error in front of you as you try out each path in your application to discover which permissions a `Subject` must be granted. This approach may work for small applications, or applications that depend on few third party libraries, but will be cost prohibitive for medium to large applications.

If you're concerned about securing your application to the hilt, going through this process may be worth it to you. Otherwise, should strongly consider not using the `DoAsPrivileged` filter, and instead performing security checks in the sensitive parts of your code, such as the code that retrieves or updates sensitive data.

## *10.3 The Permission Tag Libraries*

With the request wrapped in a privileged block, we can now use JAAS authorization methods, namely `AccessController.checkPermission()`. As our example, we'll create two custom tag libraries that perform two very useful functions:

- The `perm:granted` tag which will only display its body if the logged in user has been granted the permission.
- The `perm:notGranted` tag which will only display its body if the user hasn't been granted the permission.

For example, the following JSP page will display different text if the user has been granted the permission to read and write the file /tmp/test.txt:

```
<%@ taglib uri="perm-tags" prefix="perm" %>
<html>
<head><title>Permission Check</title></head>
<body>
<perm:granted type="java.io.FilePermission"
              name="/tmp/test.txt"
              actions="read,write">
Granted FilePermission to read and write to /tmp/test.txt
</perm:granted>
```

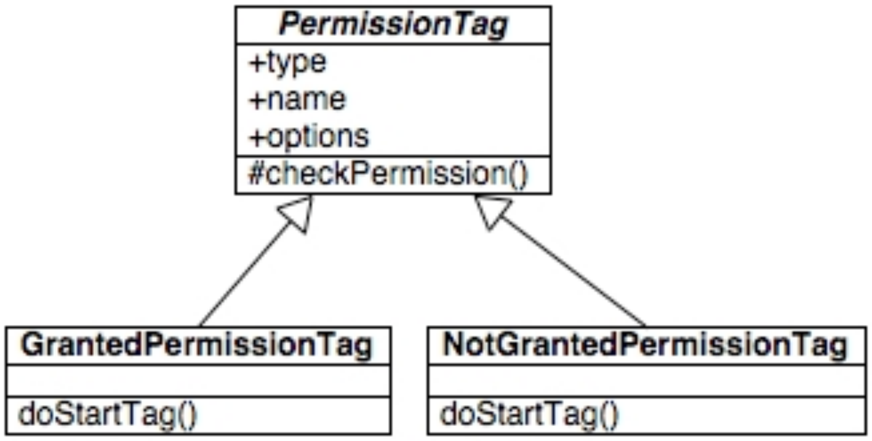
```
<perm:notGranted type="java.io.FilePermission"
                 name="/tmp/test.txt"
                 actions="read,write">
Not granted FilePermission to read and write to /tmp/test.txt
</perm:notGranted>
</body>
</html>
```

If the authenticated Subject can read and write the file, they'll see the first block of text. If they cannot read and write the file, they'll see the second block of text.

### *10.3.1 Permission Tag Classes*

The implementation of the two permission tags, perm:granted and perm:notGranted, is provided largely by the abstract class `PermissionTag`. This class has the responsibility of collecting the tag attributed, instantiating the permission with the appropriate name and optional actions, and then checking if the `Subject` has been granted the permission. The two sub-classes `GrantedPermissionTag` and `NotGrantedPermissionTag` implement `doStartTag()`, calling `PermissionTag`'s `checkPermission()`, and then displaying the tag's body accordingly.

The relationship between these 3 classes is diagramed below:



The code for each tag is listed in the following sections.

#### *PermissionTag*

The primary work done by the `PermissionTag` is done in the `checkPermission()` method. This method attempts to reflectively instantiate the permission specified by the type, name, and optional actions tags. Once the permission instance is created, it uses that instance to call `AccessController.checkPermission()`.

```
package chp10;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.security.AccessController;
import java.security.Permission;

import javax.security.auth.Subject;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;

public abstract class PermissionTag
    extends TagSupport {

  protected boolean checkPermission() throws JspException {
    Subject ctxSubject = Subject.getSubject(AccessController
        .getContext());
    String type = getType();
    if (type == null) {
      throw new NullPointerException("type is null.");
    }

    String name = getName();
    String actions = getActions();
    Permission perm = null;

    Class clazz = null;
    try {
      clazz = Class.forName(type);
    } catch (ClassNotFoundException e) {
      throw new JspException(type + " was not found.", e);
    }

    if (!Permission.class.isAssignableFrom(clazz)) {
      throw new IllegalArgumentException(type
          + " is not a java.security.Permission.");
    }

    try {
      if (name != null && actions == null) {
        Constructor c = clazz
            .getConstructor(new Class[] { String.class });
        perm = (Permission) c.newInstance(new Object[] { name });
      } else if (name != null && actions != null) {
        // name and actions
        Constructor c = clazz.getConstructor(new Class[] {
            String.class, String.class });
        perm = (Permission) c.newInstance(new Object[] { name,
            actions });
      } else {
        throw new NullPointerException(
            "Permission name must be specified.");
      }
    } catch (SecurityException e) {
      throw new JspException(e);
    } catch (NoSuchMethodException e) {
      throw new JspException("Could not instantiate " + type
          + " instance.", e);
    } catch (IllegalArgumentException e) {
      throw new JspException(e);
    } catch (InstantiationException e) {
      throw new JspException(e);
    } catch (IllegalAccessException e) {
      throw new JspException(e);
    } catch (InvocationTargetException e) {
      throw new JspException(e);
    }

    boolean granted = true;

    try {
      AccessController.checkPermission(perm);
    } catch (SecurityException e) {
      granted = false;
    }
    return granted;
  }

  public String getActions() {
    return actions_;
  }

  public void setActions(String actions) {
    actions_ = actions;
  }

  public String getName() {
    return name_;
  }

  public void setName(String name) {
    name_ = name;
  }

  public String getType() {
    return type_;
  }

  public void setType(String type) {
    type_ = type;
  }

  private String type_;
  private String name_;
  private String actions_;
}
```

*GrantedPermissionTag*

The `GrantedPermissionTag` displays the tag body if the permission has been granted, or omits the tag's body if the `Permission` has not been granted:

```
package chp10;

import javax.servlet.jsp.JspException;

public class GrantedPermissionTag
    extends PermissionTag {
  public int doStartTag() throws JspException {
    boolean granted = checkPermission();

    if (granted) {
      return EVAL_BODY_INCLUDE;
    } else {
      return SKIP_BODY;
    }
  }
}
```

### *NotGrantedPermissionTag*

The `NotGrantedPermissionTag` displays the tag body if the permission is not granted, or omits the body if the permission has been granted:

```
package chp10;

import javax.servlet.jsp.JspException;

public class GrantedPermissionTag
    extends PermissionTag {
  public int doStartTag() throws JspException {
    boolean granted = checkPermission();

    if (granted) {
      return EVAL_BODY_INCLUDE;
    } else {
      return SKIP_BODY;
    }
  }
}
```

### *Tag Library Descriptor*

The following TLD is used to specify the usage of the two tags:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>auth</shortname>
  <uri>/WEB-INF/auth-tags.tld</uri>
  <tag>
    <name>granted</name>
    <tagclass>chp10.GrantedPermissionTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <attribute>
      <name>type</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>name</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>actions</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
  <tag>
    <name>notGranted</name>
    <tagclass>chp10.NotGrantedPermissionTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <attribute>
      <name>type</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>name</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>actions</name>
      <required>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

## *10.4 Pulling it all Together*

To demonstrate the filters and tags introduced in this chapter, we'll expand on the web application from chapter 9. As in chapter 9, the database is populated with two users, and admin and a customer, each belonging to two different roles. The admin user is granted the permission to read and write to the file `/tmp/test.txt`, while the customer's is *not* granted that permission.

### *Deploying the Example Web Application*

To deploy the example web application, change to the source code directory and type `ant deploy-chp10`. This will seed the test database with the appropriate users and permissions, configure the web application, and deploy it to the Tomcat install.

Once you start Tomcat, you'll be able to load the example web application in your browser with the URL `http://localhost:8080/jass-book-chp10/`.

### *Adding Principals to the Users*

The example web applications first uses the the SQL inserts used in chapter 9 to setup up the database. Additionally, the following SQL is used to grant `java.io.FilePermission` to all users in the admin user group access to our test file:

```
INSERT INTO permission VALUES
('file-perm-id','java.io.FilePermission','/tmp/test.txt','read,write')

INSERT INTO principal_permission VALUES
('admin-principal-id','file-perm-id')
```

### *Adding in the Filters*

First, in chapter 10's `web.xml`, we add in the `AuthenticationFilter` and the `DoAsPrivilegedFilter`, making sure to map them to all URLs:

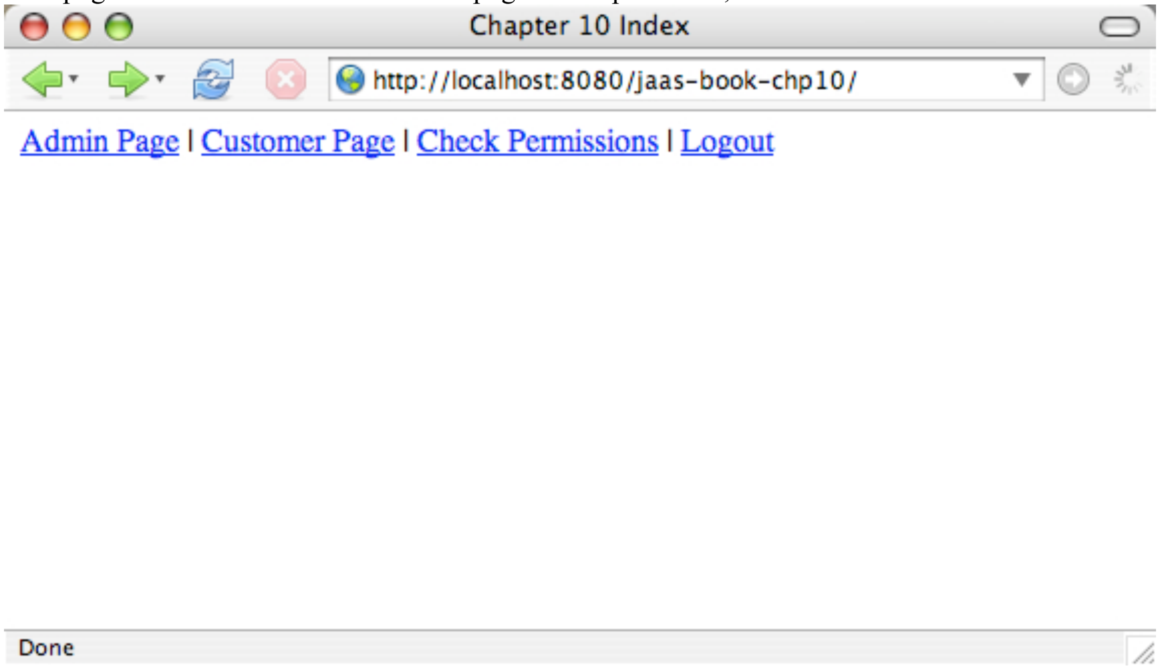
```
<!—except from src/webapp/chp10/WEB-INF/web.xml -->
<filter>
   <filter-name>authentication-filter</filter-name>
   <filter-class>chp10.AuthenticationFilter</filter-class>
   <init-param>
    <param-name>app-name</param-name>
    <param-value>chp09</param-value>
   </init-param>
   <init-param>
    <param-name>subject-key</param-name>
    <param-value>subject</param-value>
   </init-param>
  </filter>

  <filter>
   <filter-name>privileged-filter</filter-name>
   <filter-class>chp10.DoAsPrivilegedFilter</filter-class>
   <init-param>
    <param-name>subject-key</param-name>
    <param-value>subject</param-value>
   </init-param>
  </filter>

  <filter-mapping>
```

```
    <filter-name>authentication-filter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <filter-mapping>
    <filter-name>privileged-filter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

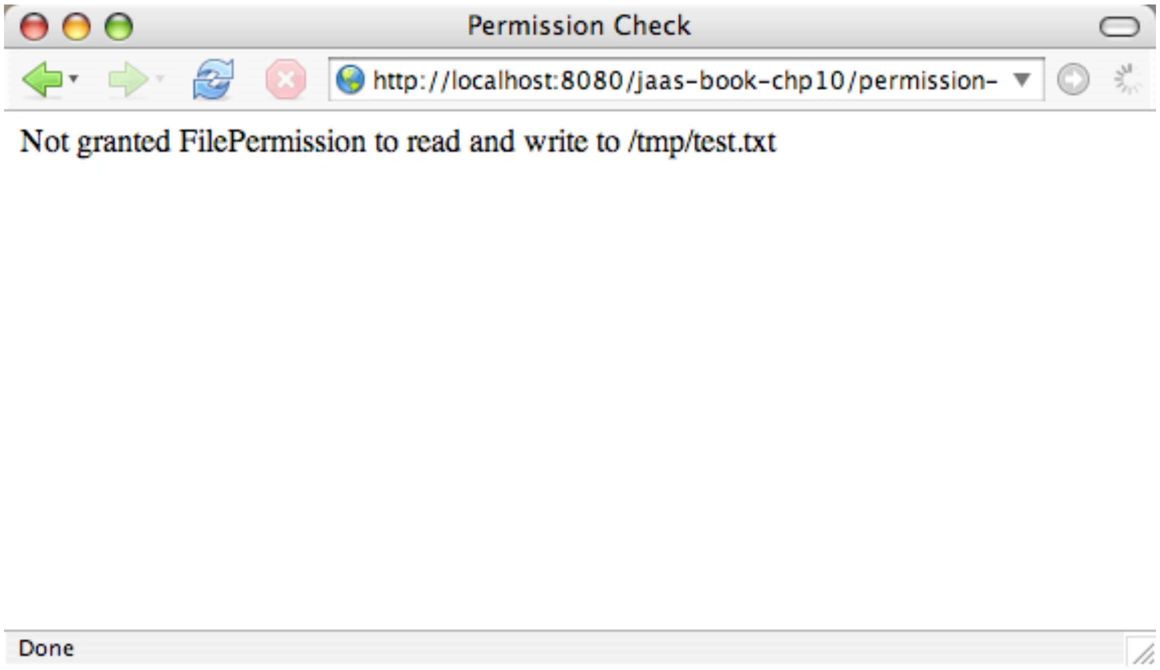
With these filters in place, we'll be able to get the currently logged in `Subject` from the session, and all requests will execute in the security context of that `Subject`.

## *Using the Taglibs*

We first add in a link to the `permission-check.jsp` from above to the index page. The first page we see looks familiar to the page in chapter nine, but has a Check Permissions link:

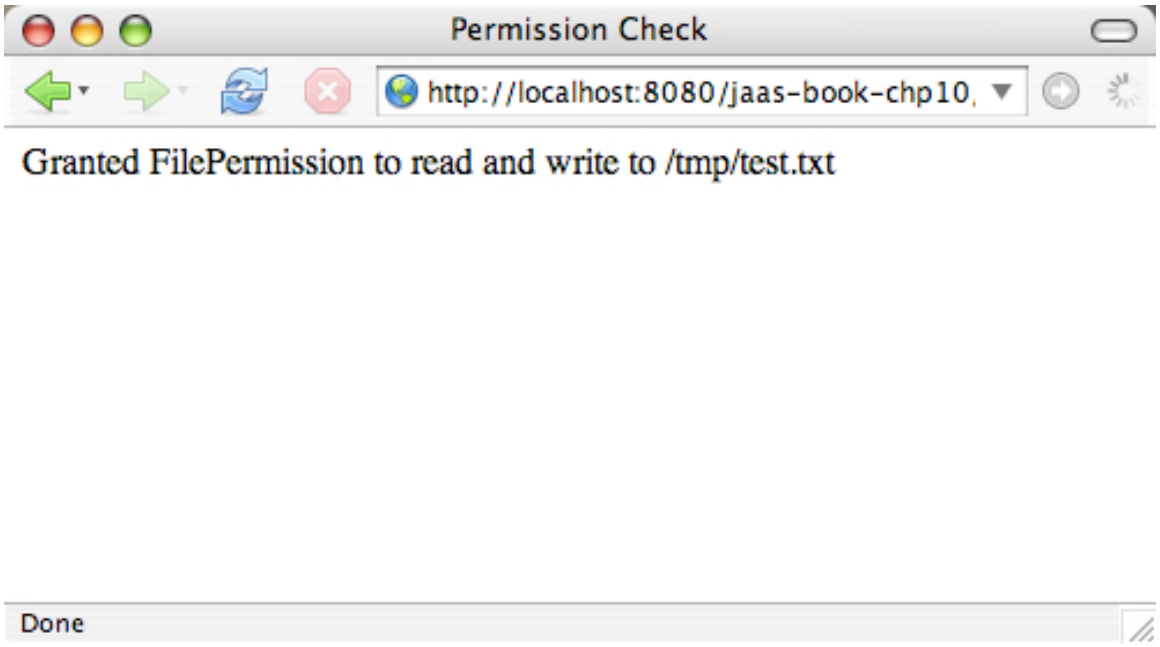

Chapter 10 Index
http://localhost:8080/jaas-book-chp10/
Admin Page | Customer Page | Check Permissions | Logout
Done

The page `permission-check.jsp`, listed above in section XXX, uses the permission tags to display different messages when the authenticated user is granted the correct `java.io.FilePermission` and when the user isn't granted the `java.io.FilePermission`. When a user that has't been granted permission to access the file clicks on Check Permissions, they see this message:



Not granted FilePermission to read and write to /tmp/test.txt

Once the user is granted permission, they'll see this message:



Granted FilePermission to read and write to /tmp/test.txt

To test this out yourself, first login as the customer with the username/password customer/secret. You'll see the first, "Not granted" page. Then, logout, and login as the admin with the username "admin" and the password "secret." This time, you'll see the "Granted" page.

## *Summary*

This chapter introduced several ways to use JAAS to secure web applications. First, we used a `ServletFilter` to get the authenticated `Subject` from the web container. Once we had the `Subject`, we could wrap an entire request in a privileged filter, allowing the code to execute as the requesting `Subject`. Finally, we implemented two tags that conditionally show their JSP bodies if the appropriate `Permission` has been granted to the logged in `Subject`. With each of the above, you can easily use JAAS to secure any web application.