

## 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 `Principal`’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`.



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

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

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());
    }
```



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

```

    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 its 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 BasePrincipals 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);
    }
}

```



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

```
}  
}
```

(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 `BasePrincipal`, we divide the two concepts into separate classes.

The code is below:



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

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

```
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 Principals...
    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;
    }
}
```



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

```
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



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

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

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.

