

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



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

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

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 `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");
```



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

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

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 `PermissionsCollection`. 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 `PermissionsCollection` 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 `PermissionsCollection`, there are two ways to obtain a concrete `PermissionsCollection` implementation: by calling `newPermissionCollection()` on a `java.security.Permission` object, or by instantiating a `java.security.Permissions` object. The `PermissionsCollection` 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 `PermissionsCollection` 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 its `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 collections¹.

java.security.Permissions

When you want to store different types of `Permissions` together, you can use the `PermissionsCollection` sub-class, `java.security.Permissions` (notice the “s” at the end). This class aggregates any number of `PermissionsCollections`, allowing a heterogeneous collection of `Permissions` to be collected together. `Permissions` provides the same behavior as other `PermissionsCollection` 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 `FilePermissionsCollection` 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



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

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

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 is a 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

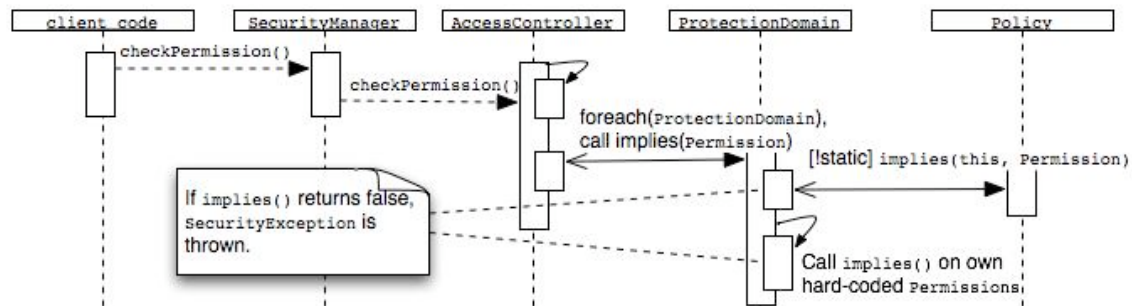
Since the first version of Java, the class java.lang.SecurityManager provides the service interface for doing all security checks. Earlier versions of Java implemented each permission check by adding a new permission checking method to the SecurityManager. This older model explains why there are so many 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.



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

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

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 below illustrates the sequence used when `SecurityManager`'s `checkPermission()` is called:



Though the `SecurityManager` delegates practically all of its 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

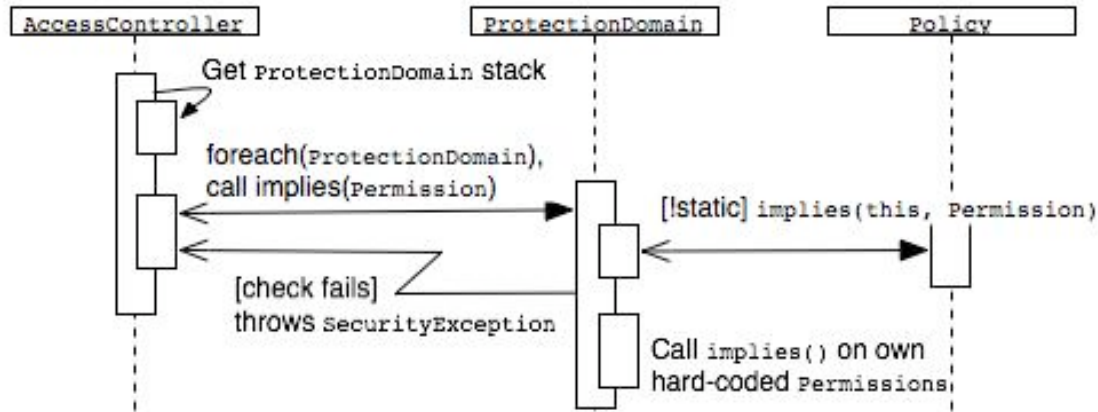
As it's name implies, the `AccessController` is at the center of JAAS. The `AccessController`'s methods fulfill three responsibilities:

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:





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 `DbPolicy`. 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.

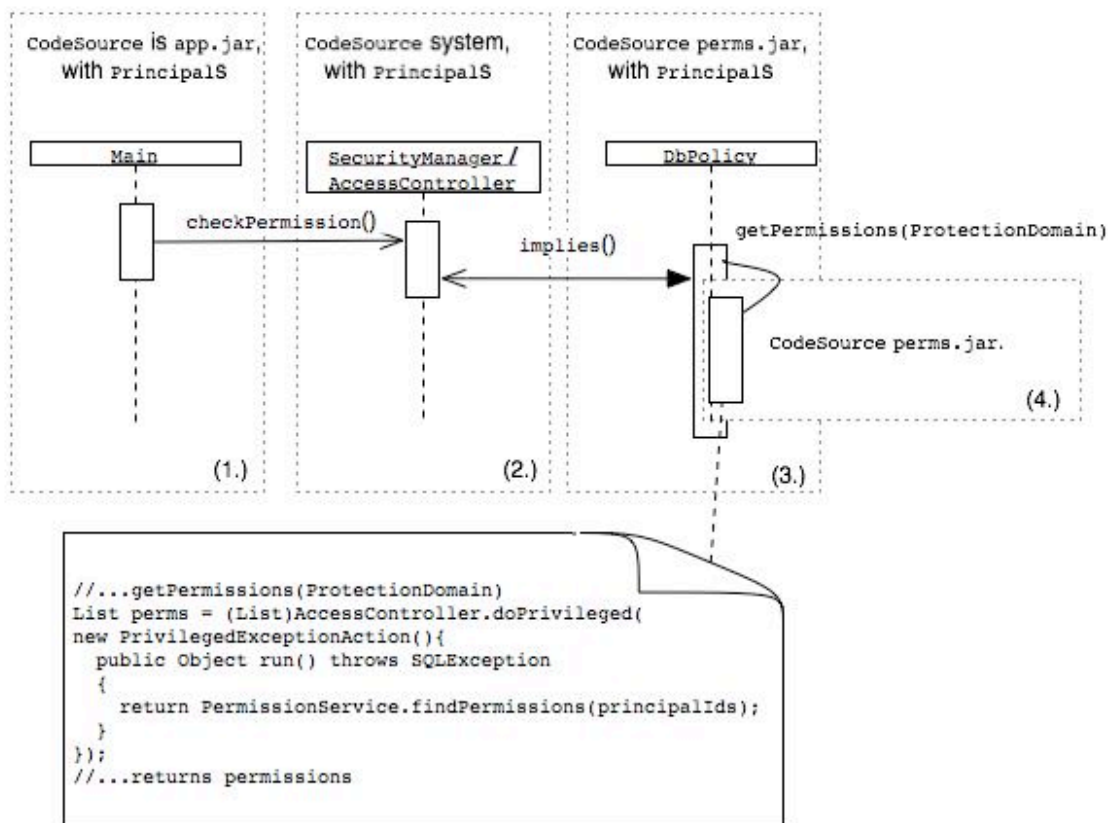


This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

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

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.



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 `AccessControllerContext` 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.doAsPrivileged

The `doAs()` methods on `Subject` provide convenience methods for creating security contexts that include the `Permissions` granted to a `Subject's Principals`. The `doAsPrivileged()` methods allows security checks to be done with *only* the `Subject's` permissions. Additionally, `AccessControllerContexts` can be optionally be passed into `doAsPrivileged()`, 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.securirty.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`



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

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

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:



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

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

- The permission classes `Permission`, `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`.

