DATABASE PROGRAMMING IN JAVA

Session: 3

Java Security

Objectives

- Explain Java security architecture
- Explain security of Java applets and applications
- Describe security policy definition for applets and application
- Explain Java Authentication and Authorization Service (JAAS)
- Explain Java Secure Sockets Extension (JSSE)

Java Security Architecture

JSA intends to secure information and services from unauthorized or unintended access.

Protects the data/services from unplanned events and natural disasters.

The term Security can be defined as a state of being free from danger or injury. The concept of security is similar to that of safety.

Evolution of Java Security

JDK 1.0

- Sandbox security model

JDK 1.1

- Signed applets

Java 2

- Protection domain and Permission classes
- Services such as:
  - Key factories
  - Keystore creation and management
  - Algorithm parameter management
  - Algorithm parameter generation
  - Certificate factories

Java SE 6

- XML digital signature API
- Smart card I/O API
- Support for AES encryption type

Java SE 7

- Support to Elliptic Curve Cryptography
- Disabling of weak cryptographic algorithms
- Enhancements to API to support Web applications

Overview of Java Security 1-2

Bytecode verifier

- Verifies whether a given series of Java bytecodes are legal or not.

Class loader

- Loads a class file associated with an application or applet, works in conjunction with `Security Manager` and `Access Controller`.

Code source

- Contains information such as author name and origin of the code for authentication.

Permissions

- Used to define the access of applications to various system resources.

Protection domains

- Various classes are grouped together to form a protection domain and permission is associated with each protection domain.

Overview of Java Security 2-2

Policy file

- Comprises the permissions defined for an application on various resources.

Security manager

- Performs appropriate checks.

Access Controller

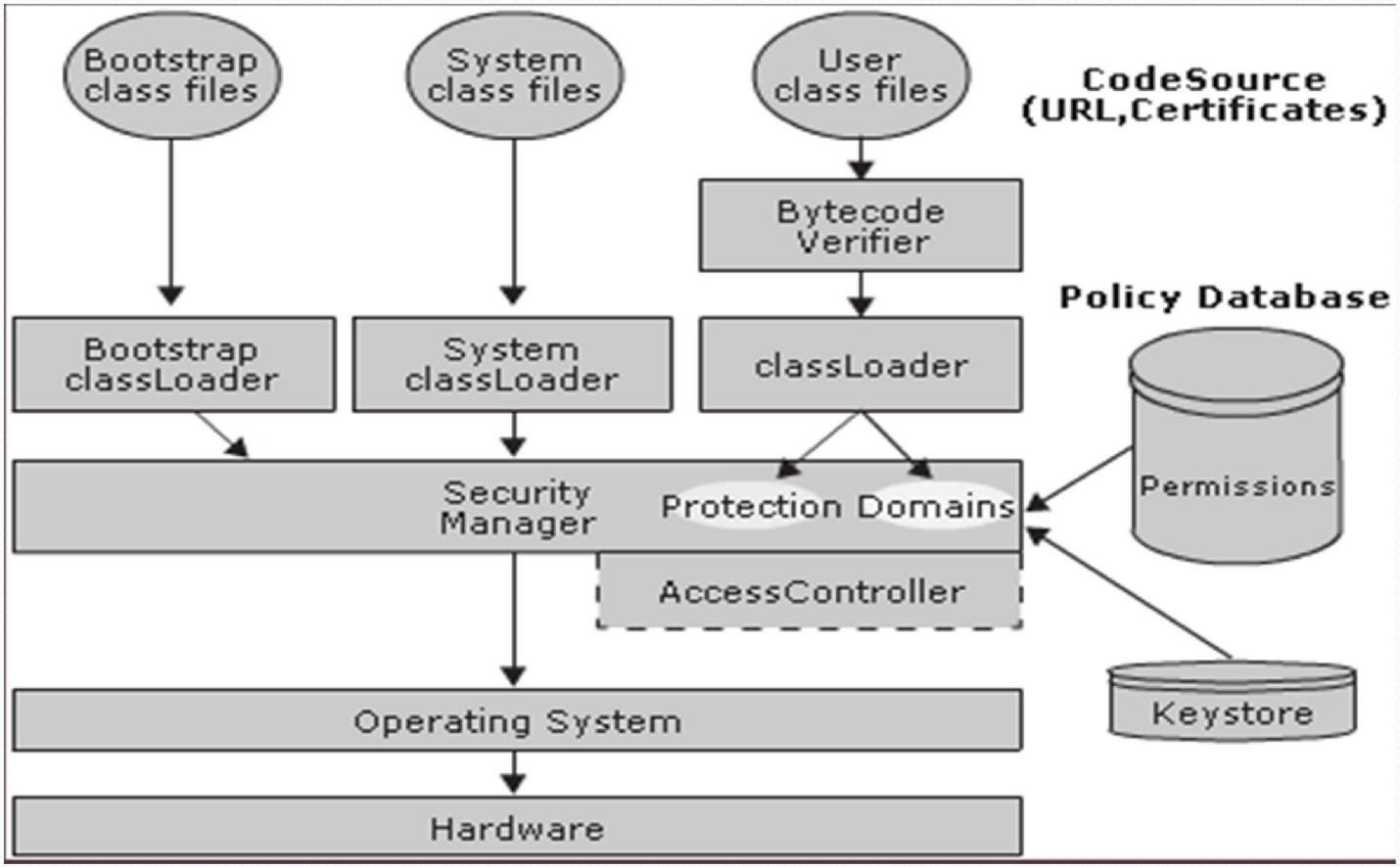
- Used to override Security Manager.

Keystore

- A password protected database comprising private keys and certificates.

Java Security Model

Following figure shows how various components of Java Security Model interact with each other:



Goals of Java Security

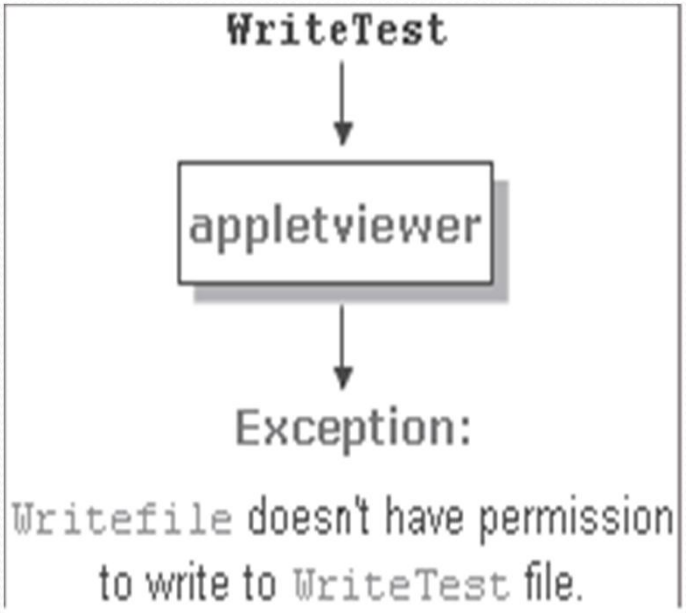
- Safety from malevolent programs
- Non intrusive
- Authenticated
- Encrypted
- Audited
- Well defined
- Verified
- Well behaved
- Object orientation and modern memory management
- Built-in access level

Securing Java Applets

Java applets are small applications written in Java that can run in an IDE on a standalone system or can be delivered over the network as bytecode or can be launched from a Web page.

Malware can be embedded in the applet code which can pose a threat to the native machine.

- Following figure shows how security policy is implemented on applets:



Types of Security Restrictions on Applets

File Access Restrictions

- Applet is not allowed to access the local file system.

Network Restrictions

- Cannot listen for incoming socket connections or datagrams, except the origin Web server.

Other Security Restrictions

- Non-local applets may not access the system properties nor define their own class loaders.
- Applets may not call native methods.

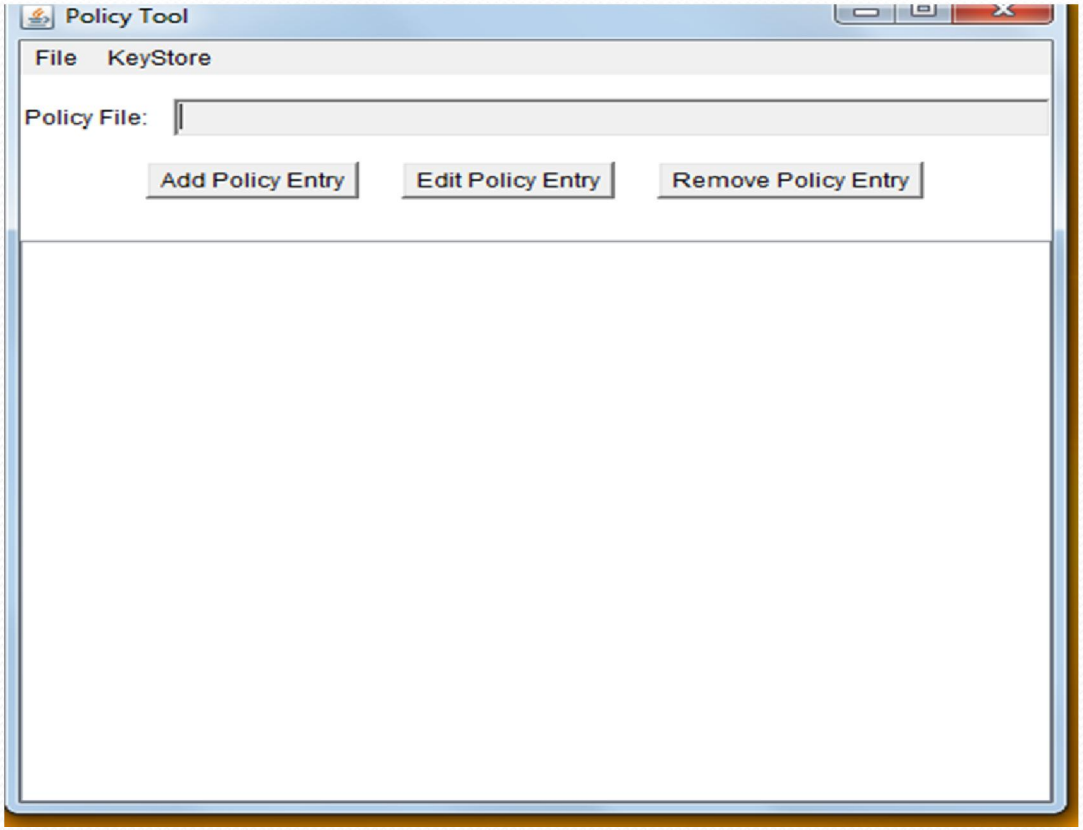
Setting Up Policy File

Following are the steps involved in creating and modifying a policy file:

1. Start the Policy Tool
2. Grant the required permission
3. Save the Policy File

Start Policy Tool

- To start Policy Tool, type the following at the command line:
  `policytool`
- Following figure shows the window that will appear on running the command:



Granting the Required Permission

- **Creating a policy file**
  - By default there is a system wide security policy defined in `java.policy` file located at the following location:

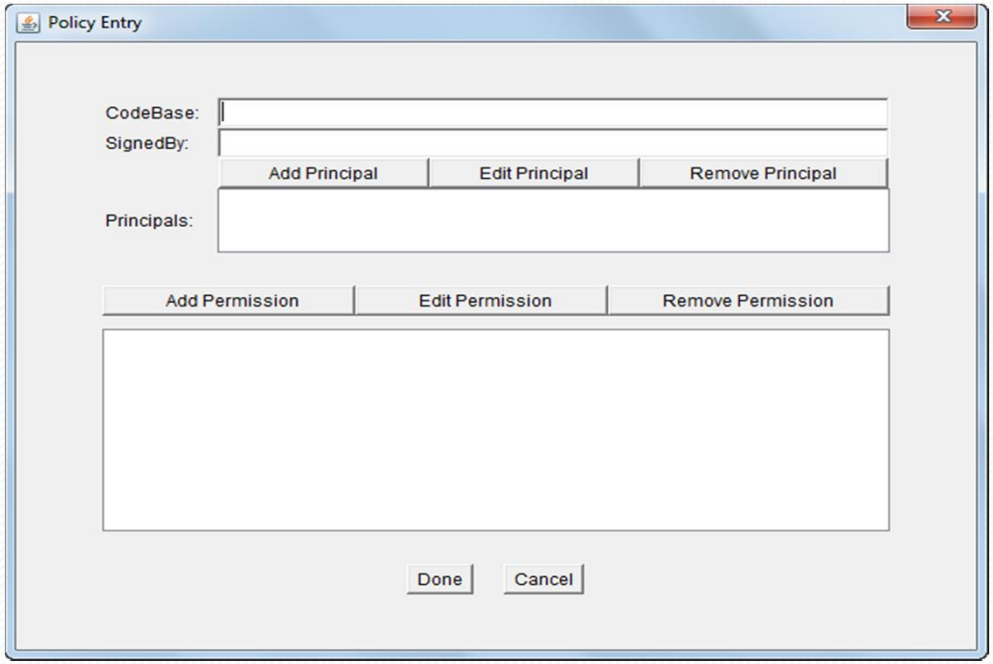
```
java.home\lib\security\java.policy
```

  - The user policy is a policy file explicitly defined by the user and is located at:

```
usr.home\.java.policy
```

Creating a New Policy Entry Through Policy Tool 1-2

- To create a new policy entry, click the **Add Policy Entry** button in the main Policy Tool window.
- Following figure shows the **Policy Entry** dialog box, when **Add Policy Entry** button is clicked:

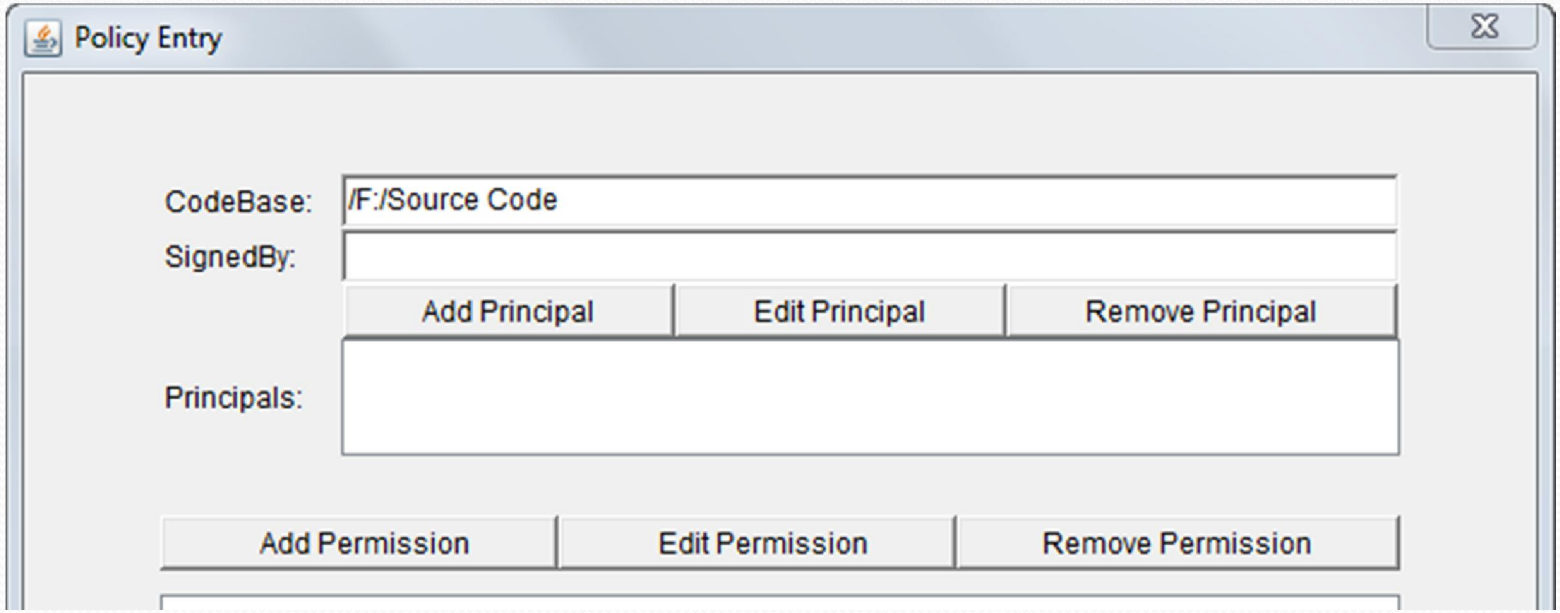


In the given screen:

- **CodeBase** value indicates the code source location.
- **SignedBy** value indicates the alias for a certificate stored in a keystore.
- **Principals** refer to the class_names that can be executed as part of the application.

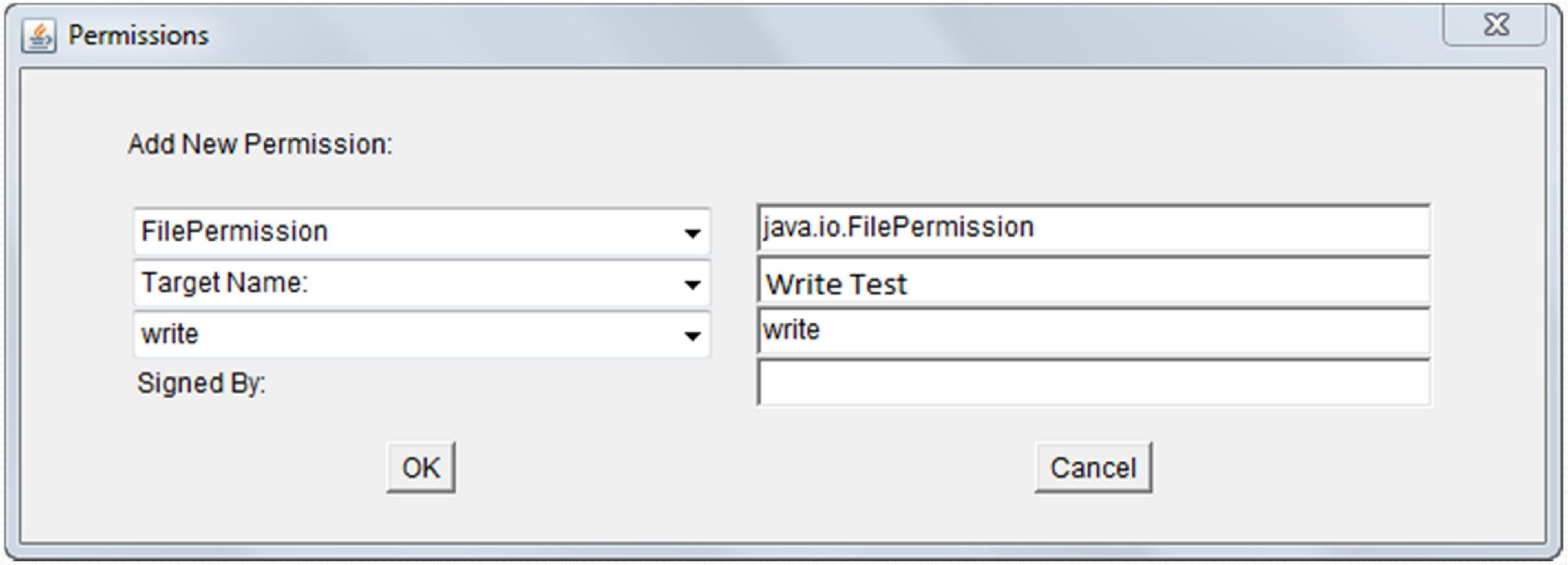
Creating a New Policy Entry Through Policy Tool 2-2

- To grant 'WriteFile' applet the permission to write to the file WriteTest, you need to type the location of WriteFile.class in the **CodeBase** box.
- In this case, the location is assumed to be file:/F:/Source Code.
- Following figure shows the entry in **CodeBase** box:



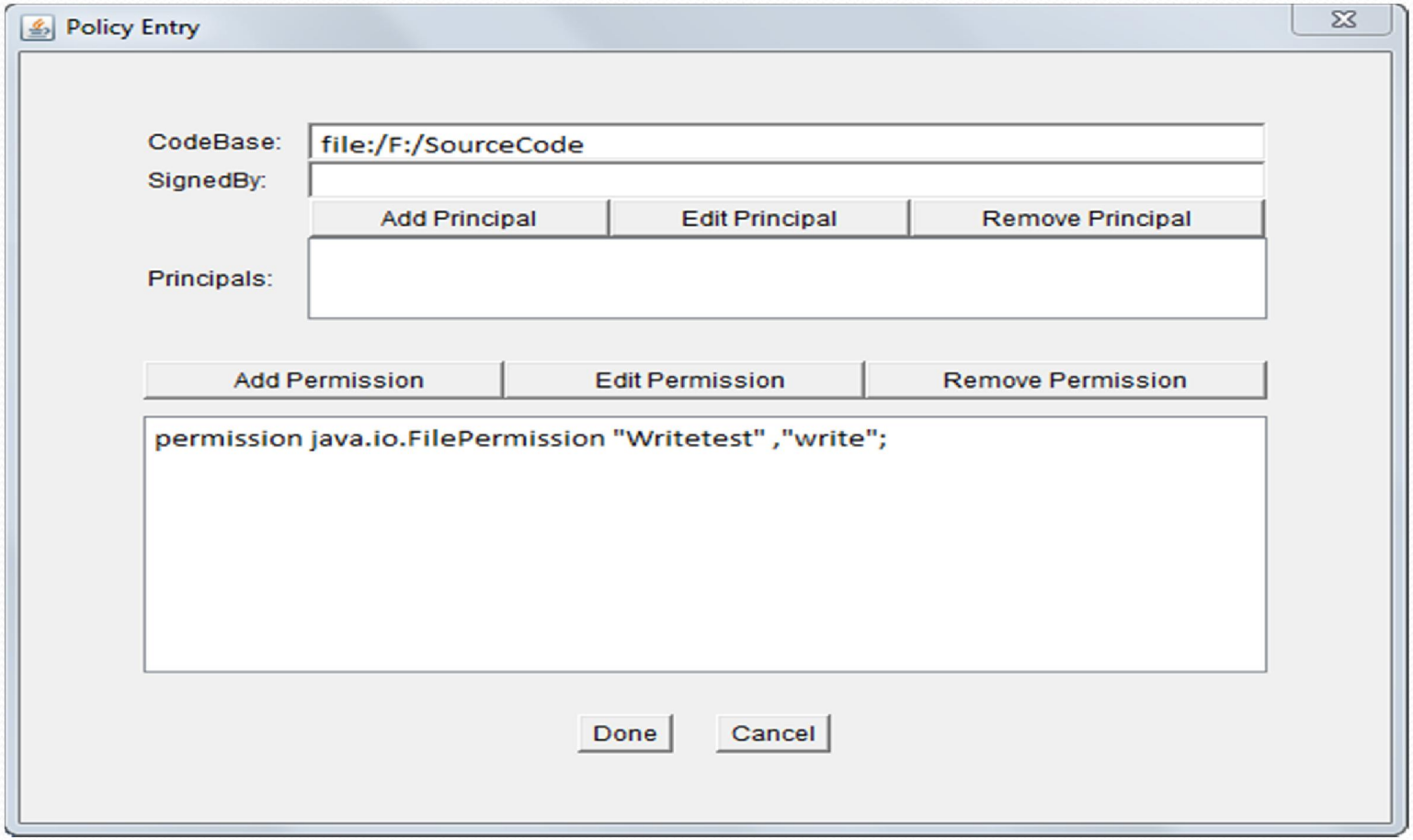
Granting the Permission

- There are three steps to grant permissions to the specified **CodeBase.**
  1. Choose the type of permission from the drop-down list.
  2. Define the target name.
  3. Specify the type of access in that third drop-down list.
- In the example, we are granting write permission on `WriteTest` file.
- Following figure shows how the **Permissions** dialog box would appear:



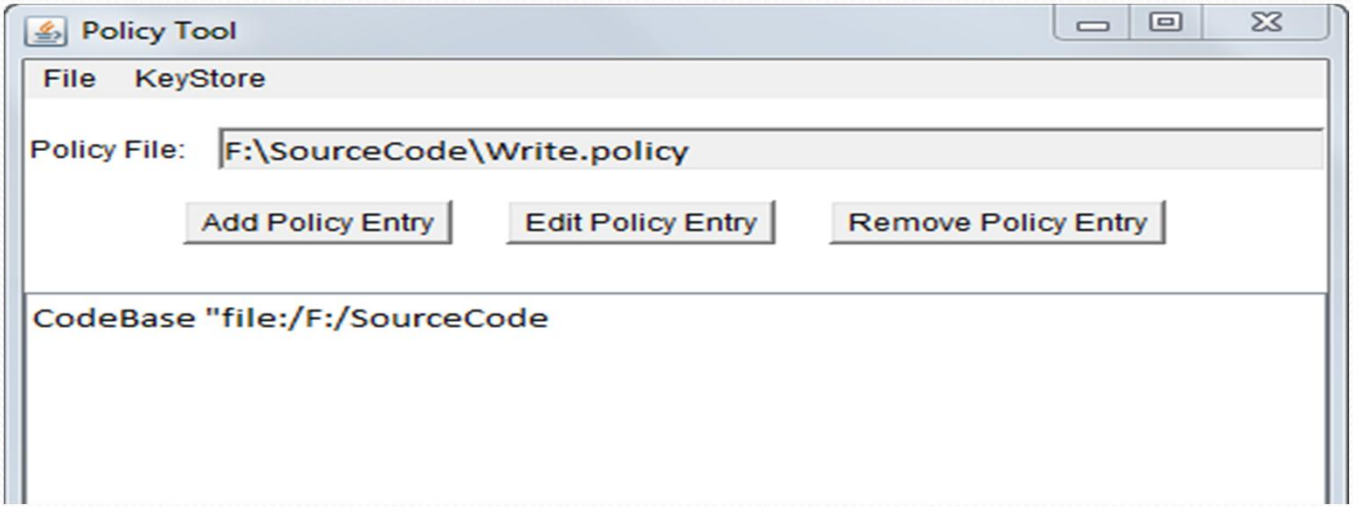
Updating the Policy Entry

- The new permission gets displayed in one line in the **Policy Entry** dialog box as shown in the figure, it can be updated by selecting the permission and editing it.



Save the Policy File

- To save the new policy file, choose the **Save As** command from the **File** menu. This displays the **Save As** dialog box.
- Save the file at the path **F:\Source Code**, named as **Write.policy**.
- Following figure shows the saved Policy File:



- To close the **Policy Tool**, select **Exit** from the **File** menu.

Policy File Effects

- Whenever an applet is run, the Security Manager invokes the default policy, `java.security`.
- An entry for a policy file takes the following form:
  - `policy.url.n=URL`
- This policy file points to various other policy files as shown in the following figure:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

Approaches to Ensure the Policy File Effects

- **Approach 1**: Specify the policy file as an argument with the appletviewer as shown in the following code:

```
appletviewer -J-Djava.security.policy=Write.policy
WriteFile.java
```

  This command assumes that the `WriteFile.class` and `Write.policy` are in the same path.

- **Approach 2**: Add a line in the `java.security` file specifying the additional policy file. The following code can be added to the `java.security` file:

```
policy.url.3=file:/F:/Source Code/Write.policy
```

Securing Java Applications

- Following mechanisms ensure that malevolent applications cannot access system resources:

JVM performs bytecode authentication.

JVM performs memory bound checks.

The Security Manager ensures execution of applications in sandboxed environment.

The Security Manager cryptographically signs java applications.

The Security Manager defines the access policy of the application.

The Security Manager is disabled by default for java applications.

Application Freedom 1-3

- The following Code Snippet considers an application named ViewProp.java trying to access system properties such as `os.name`, `java.version`, `user.home`, and `java.home`:

Code Snippet:

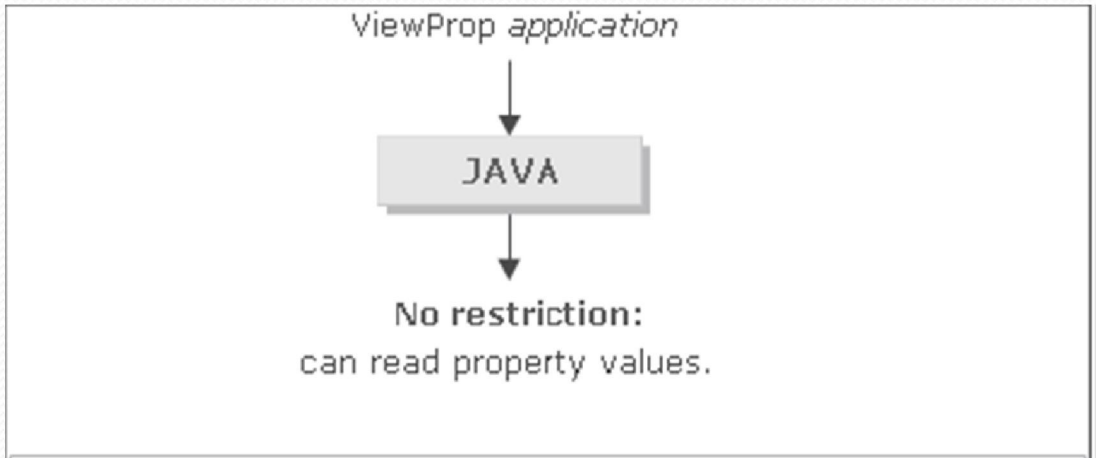
```
import java.lang.*;
import java.security.*;
class ViewProp {
public static void main(String[] args) {
/*Test reading properties with and without security manager */
String s;
try {
System.out.println("View os.name property value");
s=System.getProperty("os.name", "not specified");
System.out.println(" The name of your operating system is: " +
s);
System.out.println("View java.version property value");
```

Application Freedom 2-3

```
s=System.getProperty("java.version", "not specified");
System.out.println(" The version of the JVM you are running is: "
  + s);
System.out.println("View user.home property value");
s=System.getProperty("user.home", "not specified");
System.out.println(" Your user home directory is: " + s);
System.out.println("View java.home property value");
s=System.getProperty("java.home", "not specified");
System.out.println(" Your JRE installation directory is: " + s);
} catch (Exception e) {
System.err.println("Caught exception " + e.toString());
}
}
}
```

Application Freedom 3-3

- Following figure shows the output for the program:



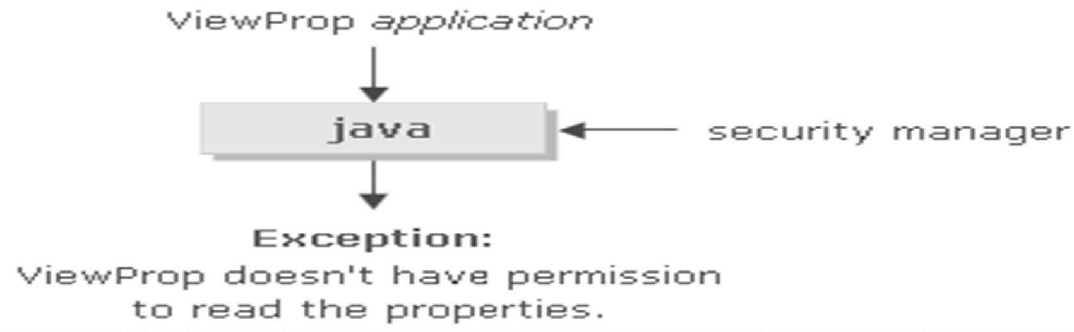

```
C:\Java\Java Examples>java ViewProp
View os.name property value
   The name of your operating system is: Window 2000
View java.version property value
   The version of the JUM you are running is: 1.5.0_04
View user.home property value
   Your user home directory is:
   C:\Documents and Settings\John
View  java.home property value
   Your JRE installtion directory is: C:\Program
Files\Java\JDK1.5.0_04\jre
```

Security Restrictions on Applications

- The Security Manager is brought into action by interpreting the java byte code with Security Manager option. Following will be the statement:
  - `java -Djava.security.manager ViewProp`

- An `AccessControlException` is thrown in such a scenario. Following figure shows the response:

```
C:\Java\Java Examples>java -Djava.security.manager ViewProp
View os.name property value
   The name of your operating system is: Window 2000
View java.version property value
   The version of the JUM you are running is: 1.5.0_04
View user.home property value
Caught exception java.security.AcrossControlException: access
denied(java.util.PropertyPermission user.home read)
```



Granting Permission

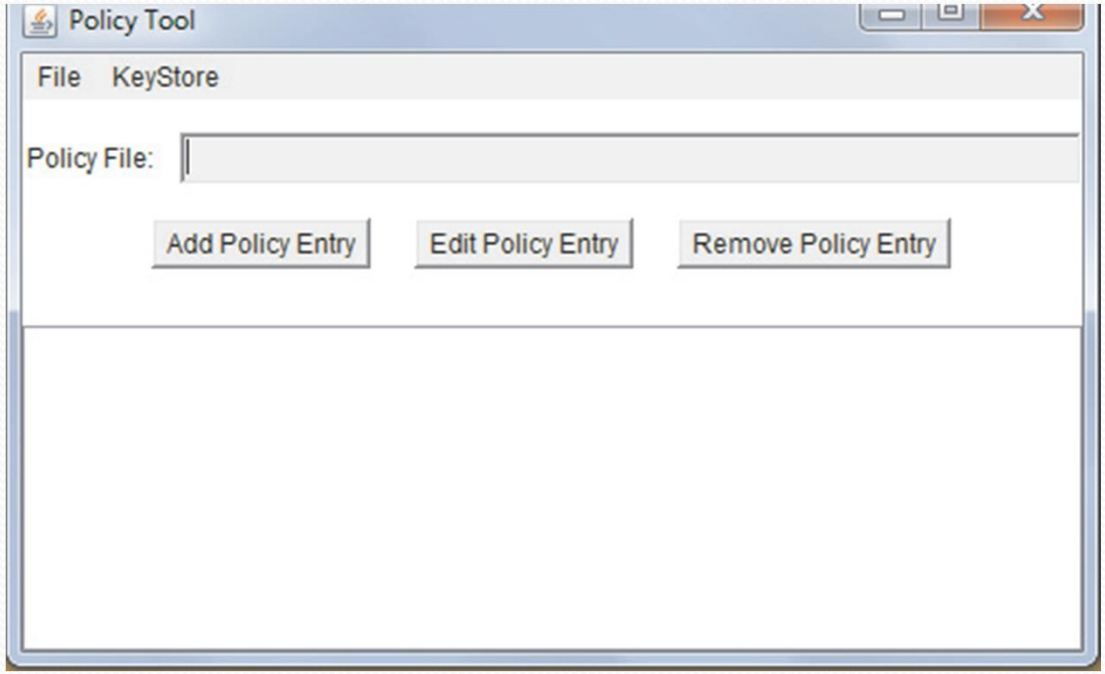
In order to grant permission a policy file has to be created. Following are the three steps to setup the policy file to grant required permissions.

- Start the policy tool
- Grant the required permissions
- Save the policy file

Following figure graphically shows the process of granting permission:

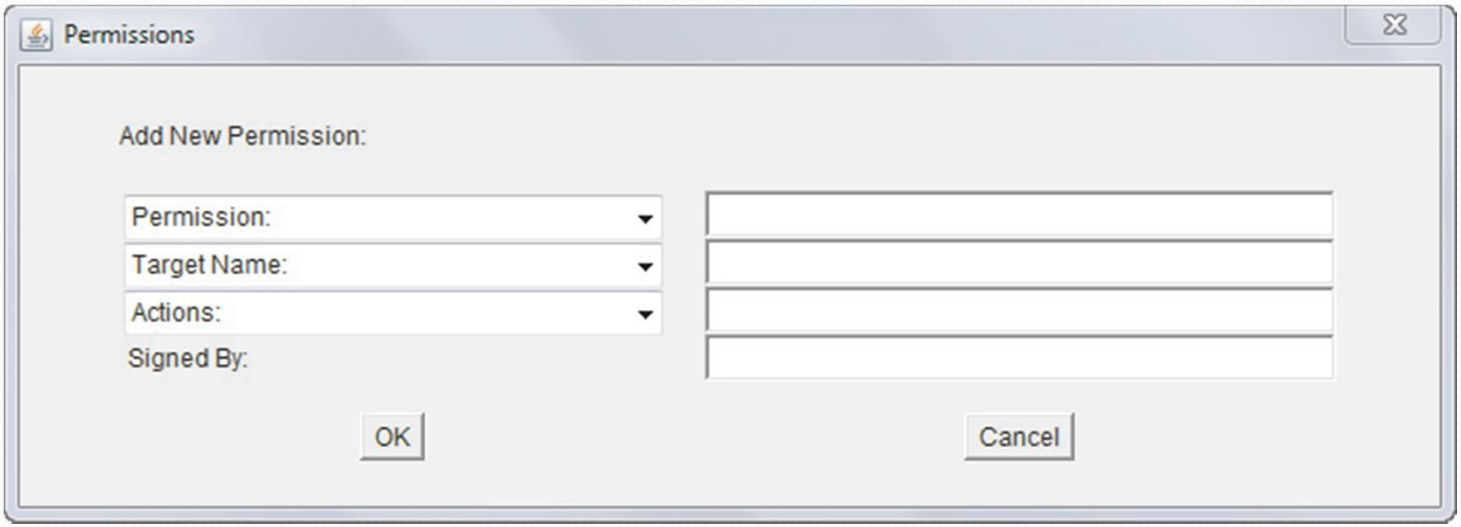
Start the Policy Tool

- The policy tool is started by executing the following command at the command line:
  - `policytool`
- This command opens the GUI policy tool, which appears as shown in the following figure:



Granting Permissions

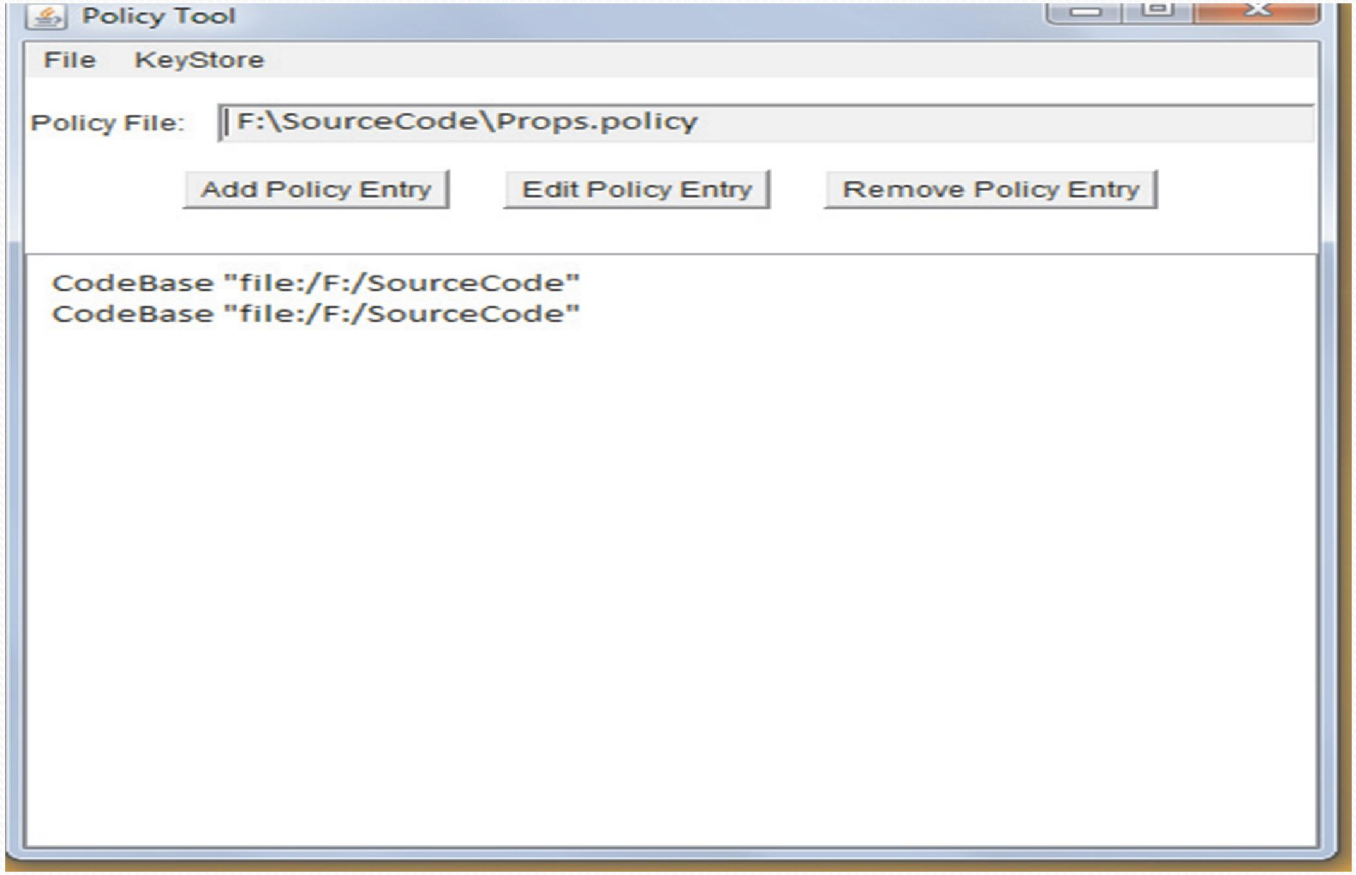
- **Adding a policy entry**
  - As in case of applets the location of the application has to filled into the **CodeBase** text box.
- **Granting the permission**
  - The values of **Permission**, **Target Name**, and **Actions** boxes are filled into the **Permissions** dialog box as shown in the following figure and applied by clicking **OK**.



The complete permission type name is `java.util.PropertyPermission`.

Saving the Policy

- The policy file can be saved using the **File** menu. Following figure shows the `Props.policy` file:



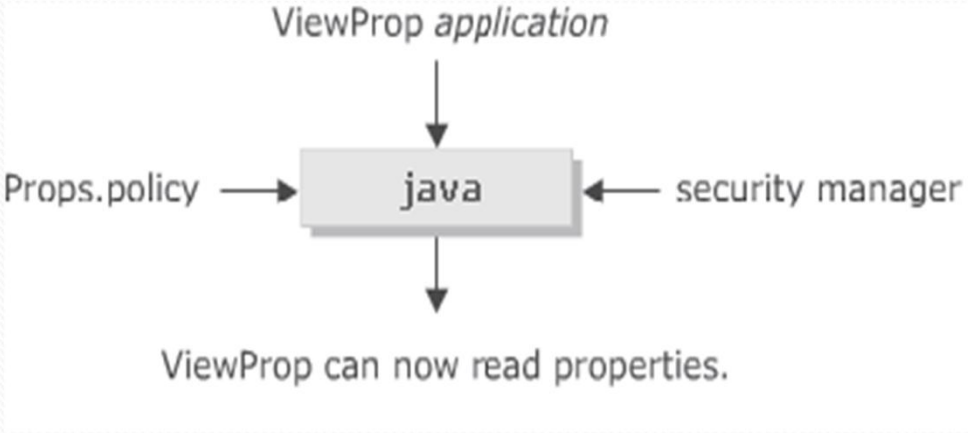
Policy File Effects

- When you run an application with a Security Manager, the policy in `java.security` file is located at the following location:

```
<java_home>\lib\security\java.security
```

- Here, `java_home` represents the location of java runtime environment.

- Following figure graphically represents the policy file effects:



Approaches to Ensure Policy File Effects

- **Approach 1**: Specify the additional policy file in a property passed to the runtime system. The following code shows how to execute the **ViewProps.java** application with the policy file:
  - `java -D java.security.manager -Djava.security.policy=Props.policy ViewProp`

- **Approach 2**: Add an entry in the `java.security` file specifying the additional policy file. The following code can be added in the default policy file to specify additional criteria:
  - `policy.url.3=file:/F:/SourceCode/Props.policy`

Granting Permissions in Java 1-3

Granting a permission in a policy file needs to specify two aspects:

- Path of class
- Permission details

Following is the syntax for granting permissions:

Syntax:

```
grant signedBy "<signer names>" codeBase "<URL>" {
/* one or more permission statements */
};
```

To grant a permission to class **Book** located in **D:\Source Code**, write the grant statement as shown in the following Code Snippet:

Code Snippet:

```
grant codeBase "file:/D:/Source Code" { /* one or more
permission statements */
};
```

Granting Permissions in Java 2-3

- Similarly, to grant permission to a class Book located over the Web, you would write the grant statement as shown in the following Code Snippet:

Code Snippet:

```
grant codeBase "http://www.mysite.com/samples/" {
/* one or more permission statements */
};
```

- A grant statement usually contains one or more permission statements. The syntax of a typical permission statement is:

Syntax:

```
permission <permission class> <target>, <action(s)>;
```

Granting Permissions in Java 3-3

- A typical policy file with the grant statement and the permission statement is shown in the following Code Snippet:

Code Snippet:

```
grant codeBase "file:/D:/Source Code" {
permission java.io.FilePermission "D:/Source
Code/data.txt", "read";
};
```

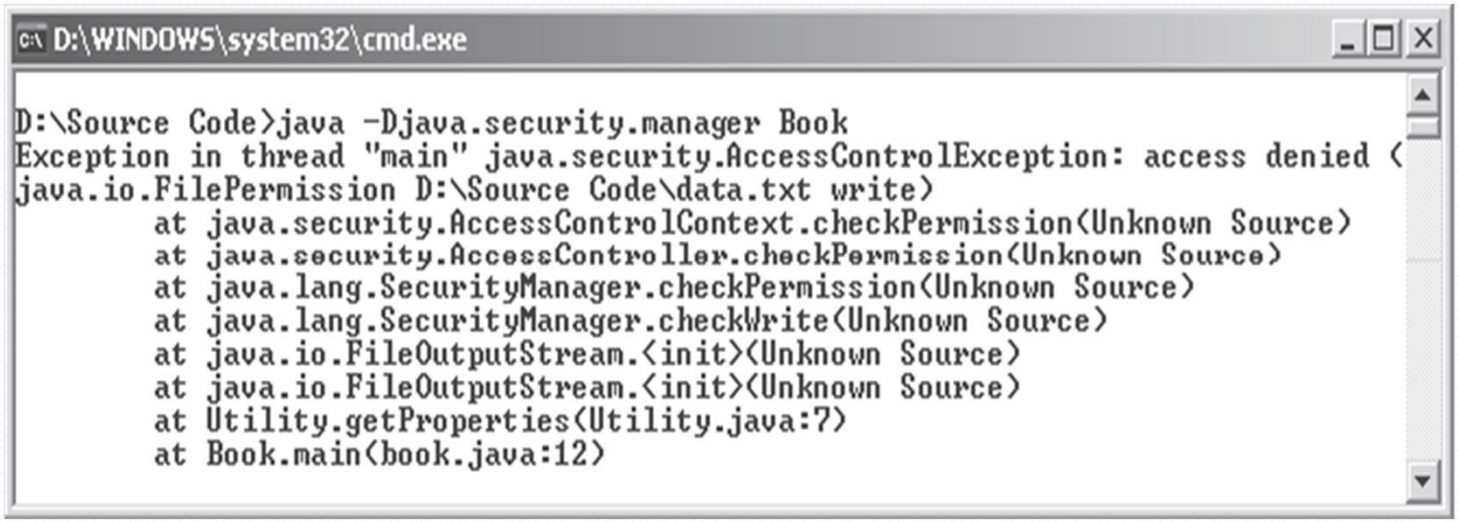
- This policy file will allow the applet or application to read the content of the file **data.txt.** The following code will not throw an exception:

Code Snippet:

```
java -D java.security.manager -D
java.security.policy=code.policy Book
// For an applet whose applet tag is specified in a .java file
itself
appletviewer -J-Djava.security.policy=code.policy Book.java
// For an applet whose applet tag is specified in a .html file
appletviewer -J-Djava.security.policy=code.policy Book.html
```

Permission Classes 1-2

- Following figure shows the `stack trace` of a program when executed with Security Manager.
- When the **`Book`** class was interpreted with an active Security Manager it generated the following stack trace as it did not have right permissions:



Permission Classes 2-2

Permission classes are used to create various permission objects which in turn are used to grant the permissions in the policy file.
Following are the various permission classes:

- `AllPermission` class
- `FilePermission` class
- `SocketPermission` class
- `PropertyPermission` class
- Other Permission classes

AllPermission Class

- The `java.security.AllPermission` class is used to allow an application or applet to access and operate on any system resource.

- Following is the code to grant `AllPermission` on the target:

```
Permission java.security.AllPermission;
```

FilePermission Class 1-2

- The `java.io.FilePermission` class is used to grant permission to operate on a file or a directory.
- Following table shows various ways of defining the targets for `FilePermission`:

Targets	Description
D:/Source Code/data.txt	Grant access to file data.txt in the path D:/Source Code.
D:/Source Code/*	Grants access to all the files in the directory Source Code.
D:/Source Code/~	Grants access to all the files in the directory Source Code and all it subdirectories.
*	Grants access to all files in the current directory.
~	Grants access to all files in the current directory and its subdirectories.
<<ALL FILES>>	Grants access to all files on the system.

FilePermission Class 2-2

- The code to define the `FilePermission` class is as follows:
  - `permission java.io.FilePermission "D:/Source Code/Addresses.txt", "read, write";`

Following are the various actions which can be defined through `FilePermission` class:

Action	Description
read	Allows to read a file or directory.
write	Allows to write to or create a file or directory.
delete	Allows to delete a file or directory.
execute	Allows to execute a file or search a directory.

SocketPermission Class 1-2

- The `java.net.SocketPermission` class is used to grant access to network resources.
- The target can be IP address, hostname, and a port.
- Following table shows various ways in which the port numbers can be specified:

Port Number	Description
5555	Port number 5555
5555-	All ports greater than 5555
-5555	All ports less than 5555

- Following table shows various actions supported by `SocketPermission` class:

Action	Description
accept	Grants ServerSocket instance the permission to accept an incoming connection from a particular host.
connect	Grants Socket instance the permission to make a connection to a host.
listen	Grants permission to create a ServerSocket instance.
resolve	Grants Socket instance the permission to check if the IP address of the host can be obtained.

SocketPermission Class 2-2

- Following Code Snippet shows permission statements using `SocketPermission` class:

Code Snippet:

```
/* Permission to connect to a system named wallpapers on port 5000.
   */
permission java.net.SocketPermission "wallpapers:5000","connect";
/* Permission to accept and listen connections on system named
   "wallpapers"
using any port > 1023 and <=65535. */
permission java.net.SocketPermission "wallpapers:1024-", "accept,
   listen";
/* Permission to accept connections from and connect to host "www.
   beautifulwallpapers.com" on port 5555. */
permission java.net.SocketPermission
   "www.beautifulwallpapers.com:5555", "connect, accept";
/* Permission to accept connections on, connect to, and listen on
   any port >
1023 and <=65535 on the localhost. */
permission java.net.SocketPermission "localhost:1024-", "accept,
   connect, listen";
```

PropertyPermission Class

The `java.util.PropertyPermission` class allows access to system properties.

The targets for this permission are the various system properties such as `java.home`, `os.name`, and `user.name`.

Following table describes the actions supported by this `PropertyPermission` class:

Action	Description
read	Allows getProperty() to be invoked on a system property.
write	Allows setProperty() to be invoked on a system property.

Other Permission Classes

Following table shows other Permission classes in Java Security:

Class Name	Description
javax.sound.sampled.AudioPermission	Allows access to lines and mixers on a system to play and record sound.
javax.security.auth.AuthPermission	Allows access to methods in java.security.auth and its subpackages.
java.awt.AWTPermission	Grants access to windowing resources such as AWT clipboard, AWT event queue, display screen to read pixels, mouse pointer information, and so on.
java.security.SecurityPermission	Grants access to sensitive information such as policy files and objects such as security provider, signer, and identity objects.
java.util.logging.LoggingPermission	Allows to control the logging configuration.
java.net.NetPermission	Allows access to specify stream handlers and network-related permissions such as the way in which authentication information is retrieved.
java.lang.reflect.ReflectPermission	Grants permissions to access fields and invokes methods in a class.
java.lang.RuntimePermission	Allows access to various runtime related resources such as class loader, Java VM, and thread.
java.io.SerializablePermission	Grants permission to subclass ObjectInputStream or ObjectOutputStream to override the default serialization or deserialization resource.
java.sql.SQLPermission	Grants permission to log the operations on the target or sync the database.

Security Manager and Permissions

Security Manager before providing access to certain resource on the system checks the policy file for permissions.

If permissions are not defined an exception is thrown.

Java Authentication and Authorization Service (JAAS)

- Independent module which handles every aspect of user authentication.
- Uses service provider approach for authentication features.
- **Authentication**
  - Authentication is the process of confirming the identity of an entity.
  - Identity is required to enforce the access control policies on the authenticated users.
- **Authorization**
  - Authorization is the process of granting or denying access to a network resource.
  - Authorization can be done through one of the following mechanisms:
    - Defining user profiles and roles.
    - Creating access control lists.

Disadvantages of Codebase Authentication

Authentication is only based on the JVM, i.e the JVM is able to execute the bytecode.

Authorization is also done on similar basis. The application running on the JVM can access all resources.

Overview of JAAS

JAAS 1.0 is a set of Java packages that enable services to authenticate and enforce access controls upon users.

Implements a Java version of Pluggable Authentication Module (PAM) framework.

Provides both authentication and authorization service.

Using JAAS for Authentication

Login Module

- Login modules for authentication are written by implementing this interface.

Login Context

- Defines the scenario when the authentication process has to begin.

Subject

- Represents the entity that requests authentication.

Principal

- Represents various attributes of Subject.

Credentials

- Information about the subject used for identifying the subject.

Authentication Process of JAAS

- An instance of `LoginContext` initiates the authentication process.
- A `LoginContext` object can be instantiated using the constructor as shown in the follows:

Syntax:

```
LoginContext(String,CallBackHandler)
```

- The `handle()` method of `CallBackHandler` invokes the `LoginModule` which in turn obtains the `Credentials` from the subject and authenticates the `Subject`.

Using JAAS for Authorization

The `doAsPrivileged()` method is invoked on an instance of `Subject`.

This method in turn calls the `run()` method.

This method contains the code to access a resource.

Java Secure Socket Extension (JSSE)

- Responsible for secure communication.
- Provides implementation of Secure Sockets Layer (SSL), Transport Layer Security (TLS), data encryption mechanisms, and data integrity mechanisms.

- **Features of JSSE**
  - It is a standard component included since JRE 1.4, thus ensuring secure communication.
  - It is implemented in Java and provides API support for SSL versions 2.0 and 3.0, TLS 1.0 and later; and an implementation of SSL 3.0 and TLS 1.0.
  - Comprises `SSLSocket`, `SSLServerSocket`, and `SSLEngine` classes that are instantiated to create secure channels.
  - Enables SSL handshaking and client server authentication.
  - Provides support for Hypertext Transfer Protocol (HTTP) encapsulated in the SSL protocol (HTTPS), that allows access to data such as Web pages using HTTPS.
  - Provides server session management APIs to manage memory-resident SSL sessions.
  - Provides support for several cryptographic algorithms commonly used in cipher suites such as RSA, RC4, DES, Triple DES, AES, Diffie-Hellman, and DSA.

Summary

- The Java 2 Security model provides a consistent and flexible policy for applets and applications. The various features of Java 2 Runtime Environment's (J2RE's) security model are Bytecode Verifier, Class Loader, CodeSource, Permissions, Protection Domains, Policy, Security Manager, Access Controller, and Keystore.
- No unsigned applet is allowed to access a resource unless the Security Manager finds that the permission has been explicitly granted in a policy file. Security restrictions vary from one browser to another.
- A Security Manager is not automatically installed when an application is running. To apply the same security policy to an application found on the local file system as to downloaded unsigned applets, you can invoke the interpreter using a command line argument.
- JAAS is the Java Authentication and Authorization Service, an API that enables Java applications to access authentication and access control services without being tied to those services. JAAS can be used for the authentication and authorization of users. In order to use JAAS authorization, the user must be first authenticated.
- Java Secure Socket Extension is used for secure communication over the Internet or any other network. It implements SSL, TLS, encryption, and so on.