



# CHAPTER 1

## J2SE SECURITY

## OBJECTIVES

*After completing “J2SE Security,” you will be able to:*

- Explain the security features built into the Java architecture, from JVM bytecode management to the Core API’s security checks.
- Describe the Java 2 security model as distinct from the traditional sandbox approach used in the earliest versions of Java.
- Use **SecurityManagers** and the **AccessController** to enforce security policies in your own applications.
- Write and configure policy files for the access controller to read.
- Create dynamic policy implementations and plug them into a running application.

# Holistic Security Practices

---

- In this course we will consider a problem that is often loosely defined – as is the course title – as “Java Security.”
- Software security can mean so many things, and can be a daunting and complex problem, especially for large systems.
- It is often decomposed into:
  - Facility
  - Hardware
  - Operating system
  - Application software
- We’re concerned with the last of these.
- For the most part we won’t worry about the first three.
  - Often we’ll assume that facilities are locked effectively, hardware is secure, and that we’re running on a modern operating system that is carefully administered to minimize improper use.
  - In some cases we’ll assume the opposite! That is, to motivate a given problem at the software level, we’ll assume that it’s possible to, say, hack the file system and place a malicious JAR in our midst.
- We will learn to assume nothing about external machines, users, and executable code.

# Threats to the User

---

- In this chapter we'll look at the J2SE Security API and architecture, and a lot of the basic techniques in this chapter go to protecting the user (and the user's files).
- Code-level security as practiced in Java would not be very important, if users only ran standalone, desktop applications.
  - The user would be responsible for excluding “malware” from his or her system, and so perhaps digital signatures on runnable code would matter. (We'll look at this practice in a later chapter.)
- But we don't just run standalone applications, particularly in offices: we run distributed applications, of many component types:
  - Applets
  - RMI classes
  - Plug-in architectures
  - J2EE components such as servlets, EJBs, and Web services
- With the many advantages of pluggable code come new threats
  - can one component misuse another component to ...
  - Destroy data?
  - Tie up the machine?
  - Tie up the network?
  - Steal information?
  - Steal passwords?

# The Java Security Architecture

---

- Java was designed from the start with security in mind.
- The language itself, and the broader executable architecture, both weave secure practices into the foundation on which application code is developed and run.
- A Java runtime is carefully layered to facilitate security at many levels:
  - The **Java virtual machine (JVM)** applies a **bytecode verifier** to provide some basic assurance about the quality of the code as soon as it is loaded into memory.
  - The JVM will only load **system classes** from a trusted source known as the **boot classpath**. (Here's one of our assumptions, then: that the party that launches a Java process has control of that path.)
  - These system classes have **exclusive access** to physical devices, and to important logical constructs such as the **file system** and **network sockets**.
  - A centralized system including the **AccessController** permits or prevents application code that wants access to those devices or to other sensitive data or services.
  - The exact behavior of that system is configurable by way of **Policy** objects, which are typically loaded from local files.

# Protections in the Java Language

---

- The Java programming language drew many lessons from other languages when it was first published.
- Ease of use and security were both served by some of these fundamental decisions:
  - Runtime **memory management**, placing the heap in the control of the JVM and its garbage collector
  - **Abstraction of object references**, again keeping JVM control of these in lieu of C-style **pointers** and the resulting chaos of “pointer arithmetic” and direct memory manipulation
  - **Strong typing**, both **primitive** and **object-oriented**, helping the programmer do what he or she wants to do in the first place by avoiding type confusion, and also making it hard for malicious code to replace a good object with a bad one at runtime
- The Java compiler enforces all of these in producing bytecode.
- But bytecode – living in .class files – then enjoys some freedom between its release from the compiler and its entry into a virtual machine.
- The bytecode verifier is there to re-assert most of these principles before the code is allowed to run.
  - It’s by no means impossible to hack .class files at the bytecode level, and anyone can simply replace one .class file with another before the JVM is launched.
  - But the verifier assures at least **internal consistency** in the class.

- In **Examples/Bytecode**, there is a simple Java application built by the White Knight corporation.
- Build and run to observe the useful functionality of the code:

```
build
run
Hello, Java!
```

- Let's say Black Queen Enterprises wants to hack this class before it runs on some client's machine.
  - See **com/blackqueen/MaliciousTweaks.java**, which parses the original bytecode and creates various modified copies.
  - Run this class using the prepared script **hack**, and observe the build directory:

```
hack
dir build\com\whiteknight
BadClassName.class
BadGreeting.class
LongerClassName.class
LongerGreeting.class
Original.class
```

- Here's an example of what **LongerGreeting.class** looks like in a binary/hex editor:

```
90: 1901000C 57656C6C ....Well
98: 2048656C 6C6F2C20 Hello,
A0: 4A617661 2107001A Java!...
```

- The **runHacked** script swaps in a given class file for **Hello.class**, and then runs the imposter.

```
runHacked LongerGreeting
```

```
Exception in thread "main"
```

```
java.lang.ClassFormatError: Unknown constant tag 74  
in class file com/whiteknight/Hello
```

- So the verifier didn't catch the actual operation – injecting content into a string literal – but it did see the internal inconsistency of a string that was longer than advertised, when it started to interpret the bytes in “Java!” as other kinds of code.

- Try **BadGreeting.class**:

```
runHacked BadGreeting
```

```
Jello, Java!
```

- Apparently the bytecode verifier does not constitute a comprehensive code-security architecture ...

# The System Class Loader

---

- In Java, even the process of loading code into memory is encapsulated in a class, and that class – **java.lang.ClassLoader** – is available to application code as well.
- This could pose a threat: what if someone supplanted one or more of the system classes themselves?
  - This is one form of **code injection attack**.
  - How would you like to have to code defensively against uses of **String**?
- But the **system class loader** (which is used by default, and inevitably for your application class) refuses to load system classes except from the boot classpath.
- Also, **ClassLoader** controls its instantiation so that, although applications (and perhaps devious plugged-in components) can create their own class loaders, there is a strict parent-child hierarchy of possible class loaders, with the system loader at the root.

- In **Examples/Core** is another small application from the White Knight corporation – this one a bit more interesting.
  - See **com/whiteknight/DNA.java**, which reads and writes data files capturing results of scientific experiments.

```
public void writeToFile (String filename)
    throws IOException
{
    DataOutputStream out = new DataOutputStream
        (new BufferedOutputStream
            (new FileOutputStream (filename), 4096));

    try
    {
        out.writeUTF (name);
        out.writeLong (whenSequenced.getTime ());
        out.writeInt (codons.length ());
        ...
    }
}
```

- **Ah, but Black Queen Enterprises is up to their tricks again!**
  - They've snookered White Knight into using their "enhanced" version of **java/io/BufferedOutputStream.java**:

```
public class BufferedOutputStream
    extends FilterOutputStream
{
    ...
    public void write (int b)
        throws IOException
    {
        super.write (b);
        if (duplex != null)
            duplex.write (b);
    }
}
```

- But, even after compiling these classes together, the user is protected from this little hack:

```
build
run
Test name:
Test 1
Codons:
GAA
Results:
8
Filename:
test1.dat
```

- All packages **java.x.y.z** and **javax.x.y.z** are loadable only from the boot classpath.
- Notice that there's no exception, no fireworks: in this case it's prevention and not cure, as the system class loader just goes to the correct source when it's time to load **java.io.BufferedOutputStream**.
- Recall that it's not an error to have multiple instances of a required class on the classpath, in general.

- The user could go out of his or her way to inject this code into the JVM – and then would suffer the consequences.
  - The only difference in this run of the application is the use of the `-Xbootclasspath` switch to the `java` launch tool:

**build**

**runHacked**

```
java -Xbootclasspath:build;%JAVA_HOME%\jre\lib\
rt.jar -classpath build com.whiteknight.DNA
```

Test name:

**Test 2**

Codons:

**CCG**

Results:

**10.6**

Filename:

**Test2.dat**

- Notice that the hack succeeded this time, creating a private copy of the results in the file **Secret**:

**dir**

```
build
build.bat
com
java
run.bat
runHacked.bat
Secret
test1.dat
test2.dat
```

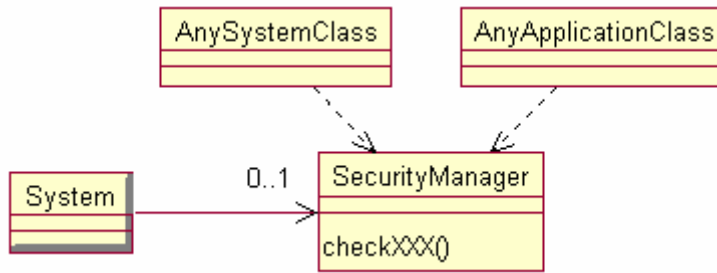
# System Classes and the Core API

---

- The Core API represents another key design decision.
- It is tempting to equate the Core API to the standard libraries in C or C++.
  - They all offer a rich body of utility functions to application code, such as Java’s Collections API, the C **strcat** function, or the C++ streams library.
- The Java Core API is more than utility, though.
- It not just a convenient way to read keyboard input or open a network socket; it is the only way.
- In other words, applications cannot get to physical devices and the operating-system API directly; they must go through the Core API, which ...
  - **Abstracts differences** between hardware and OS platforms
  - Thereby also **controls the decision-making points** for application security policies!
- If it involves the file system, windowing, networking, keyboard, mouse, or printing – and the Core API won’t do it for you – you can’t do it.
  - The only exceptions involve getting out of the JVM, for instance by way of **native code libraries**, and these do pose their own security risks.

# The Sandbox Model

---



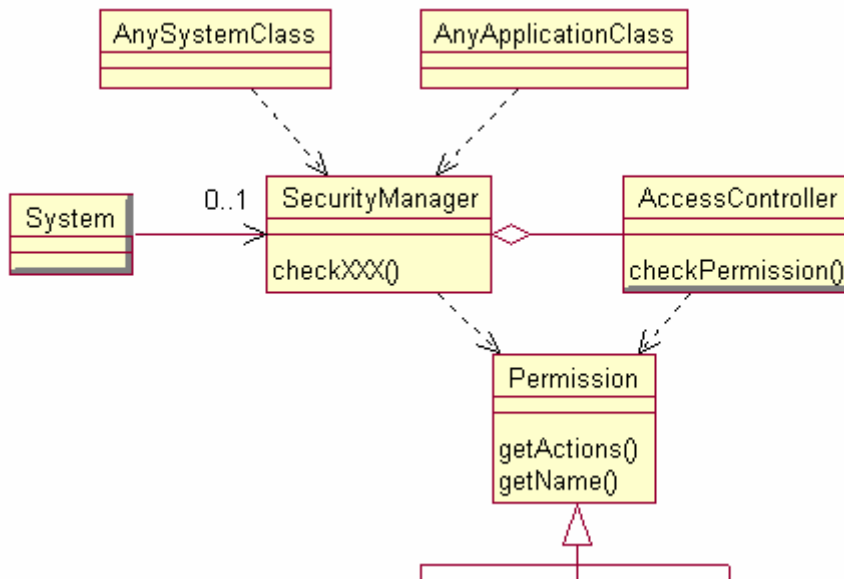
- The **system classes** (the Core API implementation plus standard extension packages such as Swing) further centralize policy-making by delegating to a single class utility called the **SecurityManager** (package **java.lang**).
- Originally, Java runtimes followed a simple **sandbox model** that recognized a binary policy:
  - So-called **trusted code** had carte blanche on the system; typically this was locally-located JARs or class files and so was presumed safe.
  - **Untrusted code** was code downloaded from a remote source – early on, this always meant an applet – and so suffered the whims of the local **SecurityManager**, which could be subclassed to effect various policies.
  - Untrusted code would then be prevented from all file operations, networking except directly to an “originating host,” reading system properties, and so on.
- Hence the term, as untrusted code was **only trusted to play in a little sandbox**.

# The Java 2 Security Model

---

- Well, it turns out that you can't do much that's useful from inside a sandbox! and generally the system was far too **coarse-grained**.
  - Really, most systems deal with a **spectrum** of users and applications, which enjoy varying degrees of trust.
  - Often it's more than a spectrum: one application might be allowed to write files but not use the network, and another might have the opposite permissions.
- “Java 2,” a.k.a. Java 1.2, brought in a new and much more sophisticated security architecture.
  - Authorization of code is expressed as a set of **permissions**, and thus can range from totally open to tightly closed.
  - Decisions are based not on applet vs. application, as any JVM may have a **SecurityManager** installed. (In fact this was true in 1.1 as well, but not widely publicized.)
  - Permissions are **granted** to **code sources**, as actors in the system, based on the location from which the code is loaded and perhaps on digital signatures provided with it.

# A New Decision-Maker



- In one of the new model's fundamental shifts, **SecurityManager** is relieved of decision-making duty.
- Now it exists as a backward-compatible adapter to the new **AccessController** class utility (package **java.security**, as most of our new classes and interfaces will be).
- **AccessController** will respond to a request from any caller for an authorization decision, by way of **checkPermission**.

```
public void checkPermission (Permission p)
    throws AccessControlException;
```

- This is nicely object-oriented, too, with **Permission** as a base for a large hierarchy of permission classes.
  - Custom permissions and therefore new types of access checks are also possible.
- Now we'll look in more detail at several of the key classes, and develop the story of the Java 2 security model as we go.

# The SecurityManager Class

---

```
public class SecurityManager
{
    public void checkAccess (Thread);
    public void checkExit (int);
    public void checkExec (String);
    public void checkRead (String);
    public void checkWrite (String);
    public void checkDelete (String);
    ...
    public void checkPermission (Permission);
    public void checkPermission (Permission, Object);
}
```

- **SecurityManager** is still the entry point for all security checks performed by system classes.
  - As such, it's the recommended entry point for application code as well, although it's feasible to skip it and to use **AccessController**, **ProtectionDomain**, or **Policy** directly.
  - It is still true that a **SecurityManager** may or may not be in play at a given time: **System.getSecurityManager** may return **null**, and only by working through this class can one be sure to adapt correctly to this contingency.
- **SecurityManager** provides individual methods for different security “checks” that code might want to perform.
  - Only some of these are listed above.
- It also offers new **checkPermission** methods that synch up with **AccessController** and its use of **Permission** objects.
  - The older **checkXXX** methods just delegate here now.

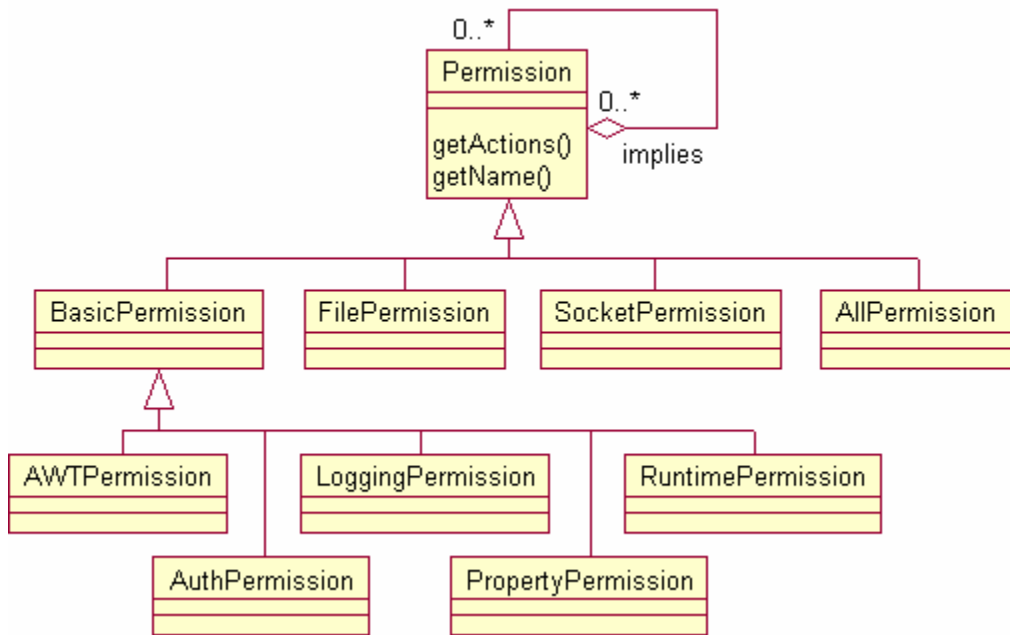
# The AccessController Class Utility

---

```
public final class AccessController
{
    public static void checkPermission (Permission)
        throws java.security.AccessControlException;
    public static native Object doPrivileged
        (... various signatures);
    public static AccessControlContext getContext ();
}
```

- **AccessController** is like that back room you've imagined, where all the decisions are really made.
  - **SecurityManager** delegates here.
  - **AccessController** uses **Thread** and **Throwable** to derive a readable stack of **Class** objects, from which it can determine the one or more **ProtectionDomains** that are in force at the time of a call to **checkPermission**.
  - It then consults whatever **Policy** is in force to see if the requested permission is granted.
- **checkPermission** either throws an **AccessControlException** or returns quietly.
  - That doesn't mean that every access failure will fall to your code in this form.
  - Unfortunately, many system classes catch this exception, and either throw one of their own or do something entirely unpredictable.

# The Permission Classes



- Each **Permission** object has a **name** – this is really just a subclassification within a particular derived class.
  - For examples: the name of a **FilePermission** indicates the file path (or class of paths, using wildcard syntax) over which permission might be granted; the name of a **SocketPermission** indicates an IP port (or range of ports).
  - Most permission types have meaningful names, but some don't: **AllPermission** needs no further qualification.
- A permission object also defines **zero to many actions** that it represents for that permission and name.
  - **FilePermission** defines “read”, “write”, “execute”, and “delete”.
  - Many permission types have no use for actions. To quote a phrase used extensively in the API documentation: “you either have the named permission or you don't.”

# Implication

---

- One **Permission** may also **imply** another.
- This is common sense for many permissions, but without understanding the meaning of the **implies** method, one can find the runtime behavior of a security policy quite confusing.
- Most implication relationships exist between permissions of the same class.
  - For instance the **FilePermission** to read all files implies a more specific permission to read file X.
- But **AllPermission** implies every other possible permission, and other inter-class implications are possible.
- System and application code does not have to do anything special to get the benefit of this feature.
  - **AccessController.checkPermission** will return successfully if the specific permission is granted, or if it is implied by any granted permission, or by an implied permission, and so on.

- In **Examples/Permissions** is a simple application that tests a list of commonly-used permissions and reports the results.
  - See **cc/security/PermissionsUtil.java**:

```
public static String reportPermission
(Permission permission)
{
    StringBuffer buffer = new StringBuffer ();

    try
    {
        AccessController.checkPermission (permission);
        buffer.append (" * ");
    }
    catch (AccessControlException ex)
    {
        buffer.append (" ");
    }
    ...
}

public static void main (String[] args)
{
    Permission[] permissions =
    {
        new java.security.AllPermission (),
        new java.awt.AWTPermission ("accessClipboard"),
        ...
    };

    for (Permission p : permissions)
        System.out.println (reportPermission (p));
}
```

- Try it out ...

**build**

**run**

```
java.security.AllPermission "<...>", "<...>"
java.awt.AWTPermission "accessClipboard", ""
java.awt.AWTPermission
    "showWindowWithoutWarningBanner", ""
java.util.logging.LoggingPermission
    "control", ""
java.io.FilePermission "-", "read"
java.io.FilePermission "-", "write"
java.util.PropertyPermission "-", "read"
java.util.PropertyPermission "-", "write"
* java.lang.RuntimePermission "exitVM", ""
  java.lang.RuntimePermission "loadLibrary.*", ""
  java.lang.RuntimePermission
    "setSecurityManager", ""
  java.net.SocketPermission
    "*", "connect,resolve"
  java.net.SocketPermission "*", "listen,resolve"
```

- **Can this be? The only thing we're allowed to do is shut down the system?**
- **Take a second look at the code, and think about this.**
  - Does this application accurately reflect the permissions given to a Java application?
  - A hint: the answer always depends on the question ...

- Our code checks permissions with the **AccessController**.
- Recall that system classes rely on **SecurityManager**, which delegates to **AccessController** – if there's a security manager in place at all!
  - If **System.getSecurityManager** returns **null**, all checks succeed, all permissions are implicitly granted.
  - If a **SecurityManager** is installed, permissions must be explicitly granted, or the application won't be able to do anything at all – no, not even shut down!
- To get a correct read of its real circumstance, **PermissionsUtil** would have to replace this line ...

```
AccessController.checkPermission (permission);
```

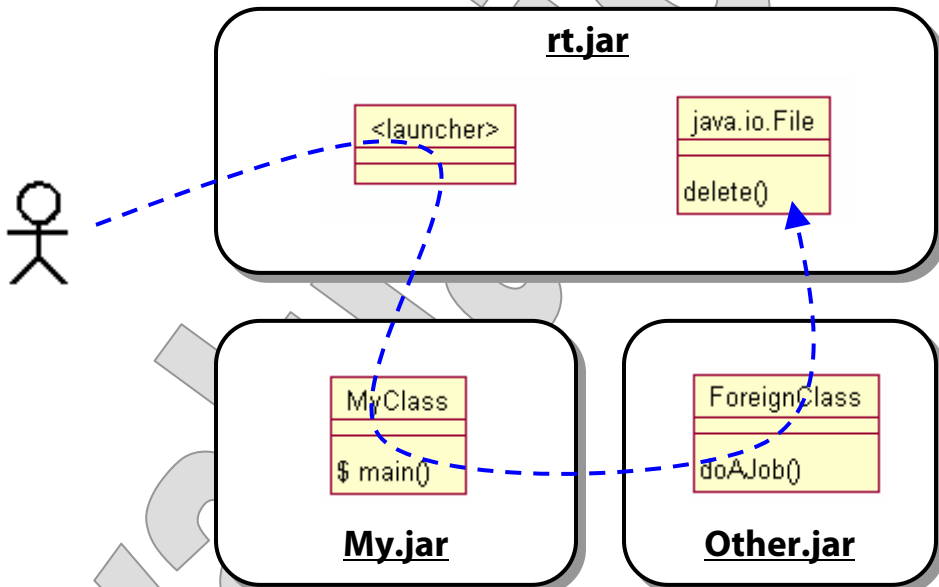
- ... with code like this:

```
SecurityManager mgr = System.getSecurityManager ();  
if (mgr != null)  
    mgr.checkPermission (permission);
```

- We won't bother to do this; since we haven't configured a **SecurityManager**, the output wouldn't be very interesting, as we'd get any permission we asked.

# Threads and ProtectionDomains

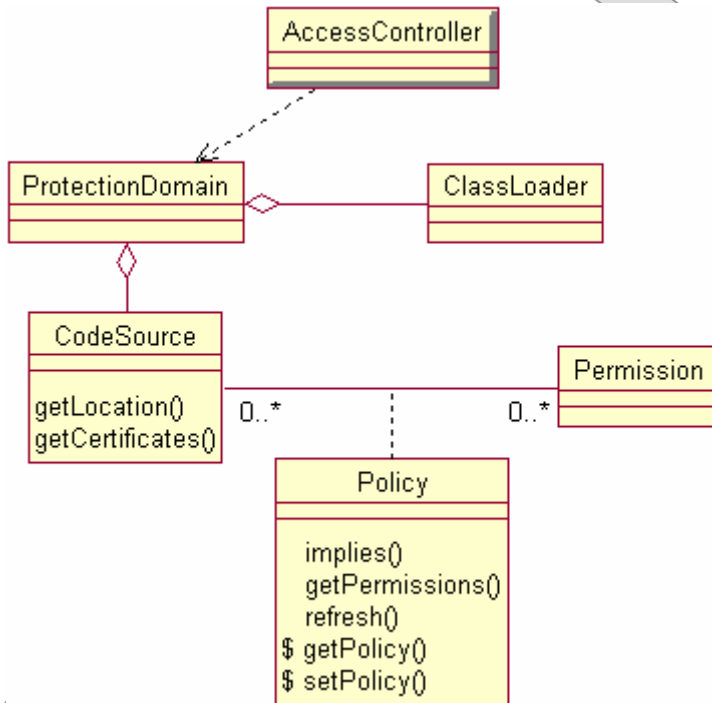
- How does **AccessController** decide which checks succeed?
  - It consults the running **Thread**, and derives an iterator over the call stack, so that it can check on each class that's been involved in directing the thread up to this point.



- Each **Class** can be queried for its **ProtectionDomain**, which in turn knows the **CodeSource**.
- Security policies in Java boil down to decisions at this level: does code from a given **CodeSource** have a certain **Permission**?

# Policies

- How does **AccessController** answer this question?
- In fact, it doesn't make that decision; it relies on a **Policy**.



- Really the **AccessController** is a processor; it's completely stateless by itself, but it knows how to carry out tasks such as parsing the call stack and interpreting policies.

# The Principle of Least Privilege

---

- So the **AccessController** can ask the **Policy** to rule on a specific permission for a specific point in the thread's execution.
- It does this for each stop along the way, and finally it applies the **principle of least privilege**, which simply says that all the participants in the thread's history up to this point must have the requested permission, or the check will fail.
- The theory behind this is that less-privileged code should not be able to take advantage of more-privileged code, either by calling it or being called by it.
- Consider a single system class – highly privileged – and a single application class – which probably enjoys fewer privileges.
  - If the application class calls a system class to do some sensitive thing, it shouldn't acquire the full permissions of the system class; it should be constrained to only those permissions granted to it.
  - Conversely, if a system class calls an application class – perhaps a callback interface supplied to a method on the system class – it shouldn't be lending its privileges to the application class from that point forward.

# The Policy Class

---

```
public abstract class Policy
{
    public static Policy getPolicy ();
    public static void setPolicy (Policy);
    public abstract PermissionCollection
        getPermissions (java.security.CodeSource);
    public PermissionCollection getPermissions
        (ProtectionDomain);
    public boolean implies
        (ProtectionDomain, Permission);
    public abstract void refresh ();
}
```

- There can only be one policy in force at one time – note that methods **getPolicy** and **setPolicy** are static.
  - Once a policy is in place, it takes a special permission to replace it – much like the security manager itself.
- A **Policy** can compute, at runtime, whether a particular piece of code may do a particular thing.
  - In the terms of the API, **Policy.implies** decides to grant or deny a specific **Permission** to a particular **ProtectionDomain**.

# The CodeSource Class

---

- If **Permissions** express the things that code might be allowed to do, it's **CodeSource** that represents the code as a distinct actor in the system that might do something.

```
public class CodeSource
{
    public final java.net.URL getLocation ();
    public final java.security.cert.Certificate[]
        getCertificates();
}
```

- Any Java class is loaded from somewhere.
  - Core API classes are uniquely privileged and don't require expression as a **CodeSource**.
  - Extension API classes are loaded from the JRE's **rt.jar**.
  - Your application is loaded from some classpath.
  - Components may come from another JAR on that same path, or be loaded from a remote source.
- **CodeSource** encapsulates the location from which code was loaded (though not the method of loading it).
- This allows for fine-grained authentication policies that mix and match specific code sources and specific permissions.
- It also carries forward any code signatures, which allows a policy to distinguish between permissions for signed vs. unsigned code.
- A **ProtectionDomain** aggregates a **CodeSource** and represents the complete and final concept of a security domain for purposes of access control.

- See the common technique for deriving the protection domain and code source at runtime, in **Examples/CodeSource**.
  - **cc/security/CodeSourceUtil.java** can get the location from which a given **Class** was loaded. Minus the error handling, this is really a simple, three-hop navigation:

```
public static URL getCodeSource (Class cls)
{
    ProtectionDomain domain =
        cls.getProtectionDomain ();
    if (domain == null)
        return null;

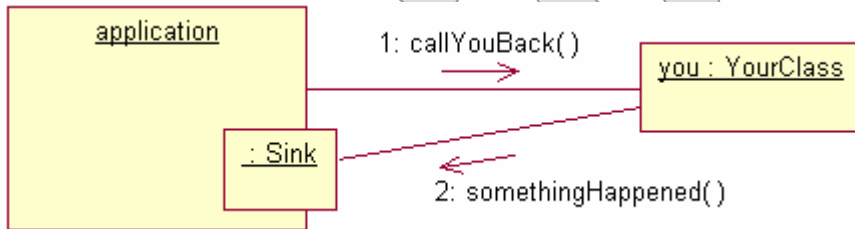
    CodeSource source = domain.getCodeSource ();
    if (source == null)
        return null;

    return source.getLocation ();
}
```

- Another utility method can list all distinct locations found in a given call stack; this is a little closer to what **AccessController** is doing when carrying out checks:

```
public static Set<URL> getCodeSources
(Throwable throwable)
{
    StackTraceElement[] stack =
throwable.getStackTrace ();
    Set<URL> result = new HashSet<URL> ();
    for (StackTraceElement element : stack)
        try
        {
            String name = element.getClassName ();
            Class cls = Class.forName (name);
            URL location = getCodeSource (cls);
            if (location != null)
                result.add (location);
        }
        catch (ClassNotFoundException ex) {}
    return result;
}
```

- **Location.java** exercises this utility, along with hypothetical application classes **MyClass** and **YourClass**.
  - The final exercise involves a call through **YourClass** to a callback handler provided by **Location** as an inner class:



- Give this example a spin:

**build**

**run**

Location of MyClass class:

file:/C:/Capstone/JavaSecurity  
/Examples/CodeSource/build/

Location of MyClass object:

file:/C:/Capstone/JavaSecurity  
/Examples/CodeSource/build/

Location of YourClass object:

file:/C:/Capstone/JavaSecurity  
/Examples/CodeSource/build/Your.jar

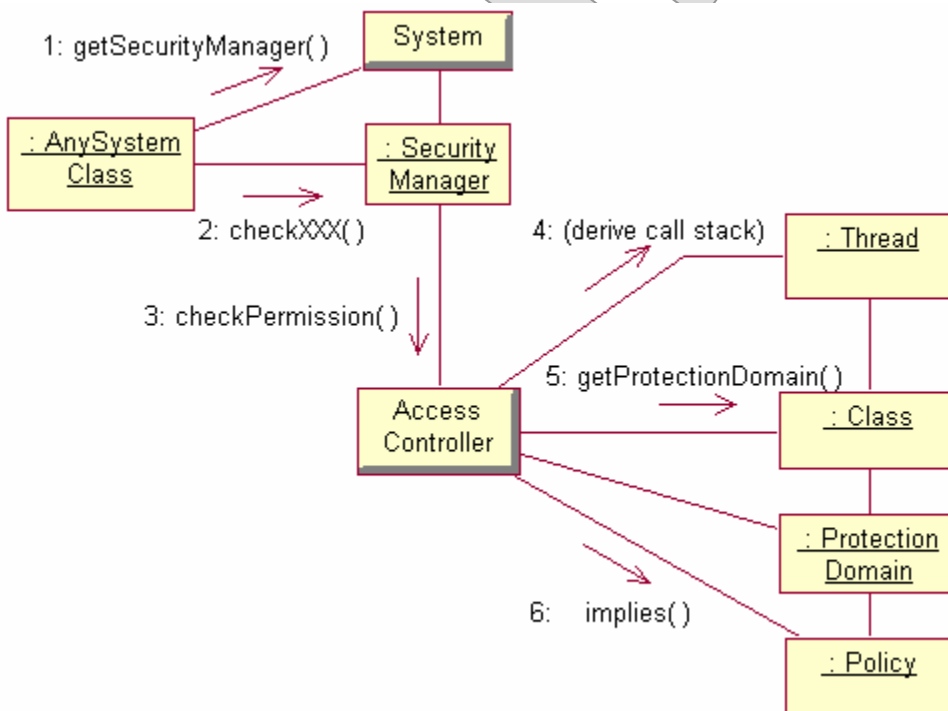
Locations of actors on callback thread:

file:/C:/Capstone/JavaSecurity  
/Examples/CodeSource/build/

file:/C:/Capstone/JavaSecurity  
/Examples/CodeSource/build/Your.jar

# Decisions, Decisions

- So, there's a **SecurityManager**, an **AccessController**, a **Policy**, some **Permissions**, a **ProtectedDomain** or two ...
- Let's review the whole process at this point. Consider the steps taken when the very first security check occurs in a new JVM – say, because someone tries to read a file.
- Here's a stab at a single, summary collaboration diagram:



- There's really just one big hole in the story we've constructed so far:
  - Application and system code makes requests for permissions.
  - Intermediate classes draw those together with policy information.
  - But how is a policy constructed?

# Configuring Java Security

---

- Thus far we've considered everything from the perspective of users of a Java API.
- In fact a major goal of the Java 2 security model is **configurability**: it should be easy to define policy choices outside of the running JVM and outside of Java code.
- Almost every choice we've seen that an application could make using its Java code can also be expressed administratively:
  - Command-line switches to the JVM
  - Configuration files
- For starters, we can decide whether a given JVM runs with a security manager or not, and which implementation to use:

```
java
java -Djava.security.manager
java -Djava.security.manager=MySecurityManagerImpl
```

- Then, there are a host of configurable features that are defined in a primary configuration file **java.security**, found in the **lib/security** directory under the JRE home.
  - This is **JAVA\_HOME** for JRE installations.
  - For JDK installations **JAVA\_HOME** is the JDK home, so the JRE home is **JAVA\_HOME/jre**.

# The java.security File

---

- **java.security** configures the complete security environment for instances of a particular Java runtime.
- Most fundamental is the choice of **provider**: the whole architecture we've been discussing is a specification, after all, with plugs for any compliant provider implementation.
- Take a look at your own **java.security** file now, and get a sense of what's defined there ...

```
...
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=
    com.sun.net.ssl.internal.ssl.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=com.sun.security.sasl.Provider
...
login.configuration.provider=
    com.sun.security.auth.login.ConfigFile
...
policy.provider=sun.security.provider.PolicyFile
...
policy.url.1=
    file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
...
```

# Policy Files

---

- Notice that there are separate providers for different subsystems of the total security architecture.
  - One of these is the provider for **JAAS**, which will be the subject of a later chapter.
  - Another is the **policy provider**, which implements the details of how a **Policy** object is incarnated from a persistent representation such as a local file.
- The reference **sun.security.provider.PolicyFile** implementation constructs a policy from one or more declared policy files.
  - At least one such file will be defined in **java.security**.
  - Applications are free to add their own to the mix ...

```
java -Djava.security.policy=My.policy ...
```

- ... or even to replace the default ones entirely ...

```
java -Djava.security.policy==MyOnly.policy ...
```

- Watch for that double-equals syntax, throughout Java security coding and configuration; it's easy to miss, and it changes things quite a bit!

# Granting Permissions

---

- The reference implementation defines a grammar for policy files; again, this is just the RI, and not a requirement of the broader J2SE specification.
- For now we'll look at the constructs required to grant permissions to a code source.
- Look at the default system policy file **java.policy**:

```
// Standard extensions get all permissions by default
grant codeBase "file:${java.home}/lib/ext/*" {
    permission java.security.AllPermission;
};
```

```
// default permissions granted to all domains
grant {
    permission java.lang.RuntimePermission
        "stopThread";
    permission java.net.SocketPermission
        "localhost:1024-", "listen";
    permission java.util.PropertyPermission
        "java.version", "read";
    ...
}
```

- A few observations:
  - A **grant** with no **codeBase** is a **grant** to all code.
  - It turns out our application code – even with a **SecurityManager** in force – does have some permissions after all!
  - It's just that none of the ones we requested were **implied** by the ones listed here.

- You may have noticed another class in **Examples/Permissions: cc.security.Exercise2**, with a **main** method that drives **PermissionsUtil** to report on a different list of permissions.
- Let's see how we do this time ...

```
java -classpath build cc.security.Exercise2
* java.lang.RuntimePermission "stopThread", ""
* java.net.SocketPermission "localhost:1024-",
    "listen,resolve"
* java.util.PropertyPermission "java.version",
    "read"
...
false
true
```

- Score! Also, see the last two lines of the output, each of which answers a question about one permission implying another:

```
Permission A = new java.util.PropertyPermission
    ("java.version", "read");
Permission B = new java.util.PropertyPermission
    ("*", "read");
System.out.println (A.implies (B));
System.out.println (B.implies (A));
```

- The one remaining mystery is why the **RuntimePermission** with the name “exitVM” is granted – run the **PermissionsUtil** application again to confirm this.
- This permission is not called out in the policy file, is it?
- What explanations can you think of that might account for these results, and how would you confirm your hypothesis?

- Perhaps one of the explicit permissions implies the “exitVM” permission.
  - We could test this by commenting out all granted permissions and running again.
  - The “exitVM” permission would still be granted!
- There are two policy files listed in **java.security**, and we’ve only reviewed one! What if the user-specific policy file grants this permission?
  - We can seek out that file and find that, for a default installation of the JDK, it doesn’t (yet) exist.
  - So this doesn’t explain it ...
- The bottom line is a little surprising: this a magic permission of sorts; it is automatically granted to all code loaded from the application classpath.
  - Find the “answer of record” in the javadocs for **java.lang.RuntimePermission**.

- Let's try an exercise in enforcing a security policy on an existing, installed application.
  - Work in **Demos/SecurityManager**.
  - The completed demo is in **Examples/SOAPSniffer/Step2**.
- 1. Run the SOAPSniffer application, which performs port forwarding for SOAP/HTTP messages and logs the traffic between a Web service and its client.

**build**

**sniff**

Forwarding local port 80 to local port 8080 ...

2. ... and Ctrl-C to break out of the process. Now, this is clearly a security issue. It may be desirable but generally a system administrator is not going to want applications to be able to forward and redirect between ports. How can we get control of the security policy over this application?

3. Run with a security manager, for starters! Edit the **sniff** script, as in:

```
java -classpath SOAPSniffer.jar  
-Djava.security.manager  
cc.tools.SOAP.SOAPSniffer %1 8080
```

4. Now try again ...

**sniff**

```
java.security.AccessControlException: access denied  
(java.io.FilePermission Traffic.txt write)
```

5. What if we want to allow this application to run, but as a matter of company policy we always run with a security manager?
6. Try editing **java.policy** to grant the necessary permission. Add the following to the **grant** for all code:

```
permission java.io.FilePermission  
    "Traffic.txt", "write";
```

7. Try again, and now we see that we are on a little journey of discovery:

**sniff**

```
java.security.AccessControlException: access denied  
(java.net.SocketPermission localhost:80  
listen,resolve)
```

8. Add this permission as shown below, and try again.

```
permission java.io.FilePermission  
    "Traffic.txt", "write";  
permission java.net.SocketPermission  
    "localhost:-", "listen";
```

**sniff**

```
Forwarding local port 80 to local port 8080 ...
```

9. That's got it. But, on further reflection, we probably don't want to grant every Java application these permissions – just SOAPSniffer as installed in a particular location. Break your two new permissions out into a separate **grant** for a specific code base:

```
grant codeBase "file:SOAPSniffer.jar"
{
    permission java.io.FilePermission
        "Traffic.txt", "write";
    permission java.net.SocketPermission
        "localhost:-", "listen";
};
```

10. You should still have success with **sniff**, but now only the standard (JDK-installed) permissions are available to all other applications.

11. A more maintainable approach is to break out application-specific grants to separate policy files, and to specify these on the command line. Cut the content above and paste it into a new file in **Demos/SecurityManager** called **Sniffing.policy**.

12. Edit **sniff** again, this time to call out the policy file:

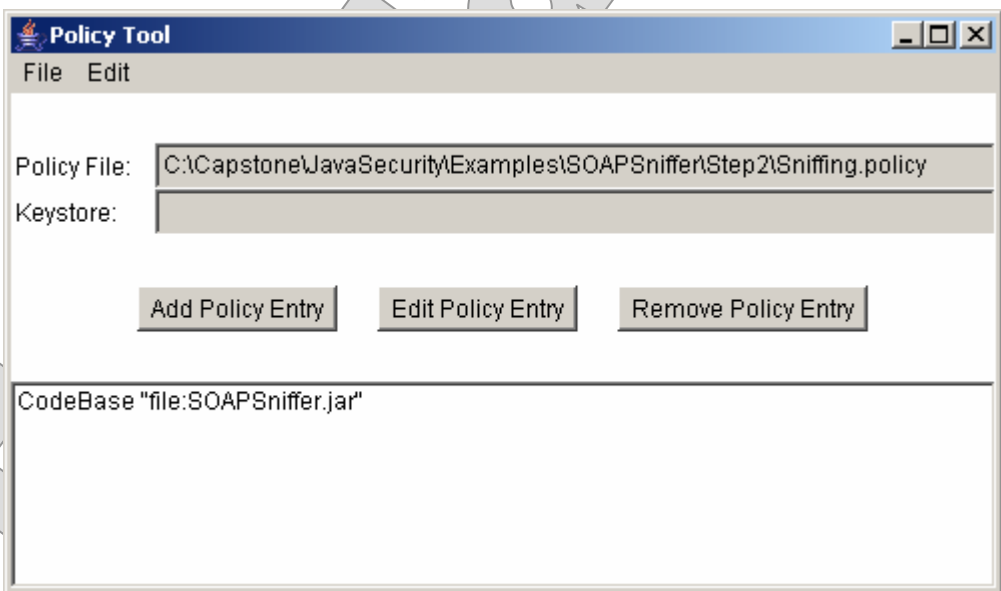
```
java -classpath SOAPSniffer.jar
-Djava.security.manager
-Djava.security.policy==Sniffing.policy
cc.tools.SOAP.SOAPSniffer %1 8080
```

13. Run one last time, again successfully.

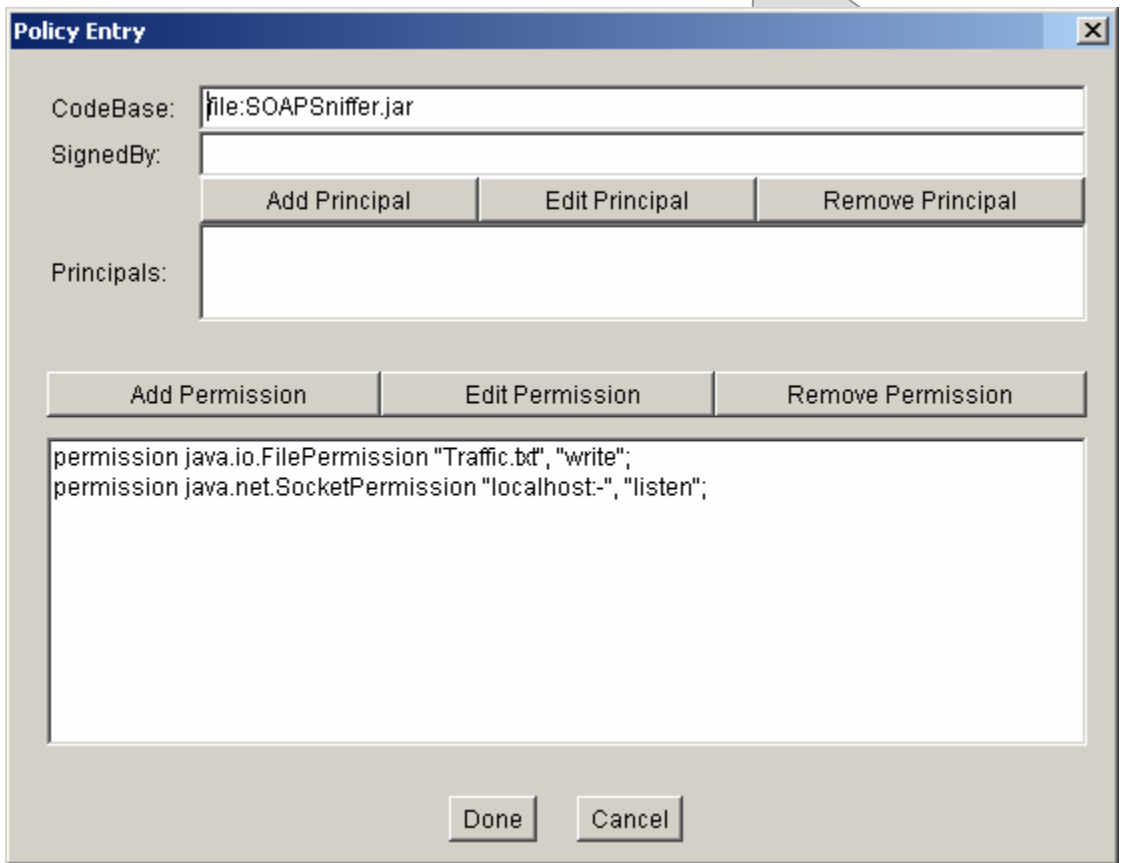
- The JDK ships with a GUI tool that helps you create and edit policy files, called **policytool**.
  - Most major Java tool vendors will provide something at least as usable as this.
- Try it out on the two files you've seen so far:

**policytool**

- **File|Open** and browse to either file – we'll look at **Sniffing.policy** here:



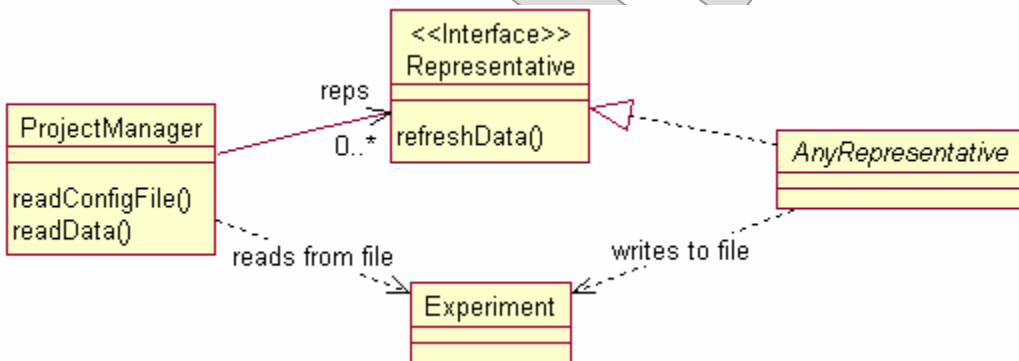
- Double-click the codebase entry:



- If nothing else, you can see that we have a good bit more to cover! In later chapters we'll discuss some of the terms shown above, including JAR signing and Principals as opposed to code bases.

## Suggested time: 30 minutes

In this lab you will add policy protection to an application that facilitates sharing of scientific data for joint research projects. The (highly hypothetical) NIH application provides a simple plug-in architecture by which participating institutions can report their ongoing work by writing files to a shared document space.



The problem is that some of these scientists are a little ... mad. And they can get awfully competitive. When you run the application, you'll observe a hack by one of them that degrades the results saved by another – probably hoping that the skewed results will result in a more favorable distribution of grant money. You'll then tighten up the application using J2SE security.

Detailed instructions are found at the end of the chapter.

# Implementing a Policy

---

- Again, what we've just seen is the reference implementation of **Policy** objects, loaded from files of a specific syntax.
- Other file formats are possible, and it's also possible to subclass **Policy** yourself.
  - Often you would aggregate another, robust implementation, and perhaps add your own diagnostic features.
  - You could start from scratch, too, and load policy contents from different sources and formats, or add and remove permissions dynamically based on some application-specific stimuli.
- Because policies are so fundamental and sensitive, Java requires that you make their classes available on the boot classpath.
  - Do everything else right, but leave your JAR on the application classpath, and the JRE will quietly pass on your offer of a policy implementation, and default to its own.
- It's also possible to install a policy programmatically, just as it is to set the security manager from your code.

- In **Demos/PlugInPolicy** is a trivial example of setting a custom **Policy** object into place on application startup.
  - This will be something like what we've done from the command line in previous exercises.
  - The completed demo is found in **Examples/Policy/Step2**.
- 1. Review **cc/security/MyPolicy.java**, which grants (i.e. **implies**) all permissions that are requested. Also there is a method **exercise** that tries to delete a prepared file, and **main** calls this method. The **run** script creates the file (**Dummy.txt**) each time, before running the application.

```
echo Useless information > Dummy.txt  
java -classpath build cc.security.MyPolicy
```

2. Run the class as an application:

```
build  
runPlugIn  
Deleted dummy.txt.
```

3. Add code to install a security manager:

```
System.setSecurityManager (new SecurityManager ());  
exercise ();
```

4. ... and test again:

```
build
```

```
runPlugIn
```

```
Exception in thread "main"
```

```
java.security.AccessControlException: access denied  
(java.io.FilePermission dummy.txt delete)
```

5. Now set the policy – better do this before the security manager's in place, or you won't be permitted to at runtime.

```
Policy.setPolicy (new MyPolicy ());
```

```
System.setSecurityManager (new SecurityManager ());  
exercise ();
```

6. Test again, and see that your policy is now in play, allowing the operation again:

```
build
```

```
runPlugIn
```

```
MyPolicy.implies() called.
```

```
Deleted dummy.txt.
```

# A Dynamic Policy Object

EXAMPLE

- If we can plug in a simple policy like that, we can branch out from there to a dynamic policy object, one that allows application code to modify it on the fly.
- See **Examples/Policy/Step3** and the **cc.security.DynamicPolicy** class, which accepts a **Map** of **CodeSource** and **PermissionCollection** objects to define its current policy.
  - This map can be updated anytime using **getPermissionsMap** and modifying the live map object.
  - Of course that in turn is a security hole! and we won't get into making this policy class industrial-grade in this example. The method should do some sort of strict check on the caller's code source, or require a secret key, or otherwise take steps to assure that those governed by the policy can't easily change it.
- Because the class manipulates policy settings (even though it ultimately does so upon itself), it requires certain **java.lang.Runtime** permissions – so it grants these to itself in the constructor:

```
public DynamicPolicy ()
{
    Permissions permissions = new Permissions ();
    permissions.add (new RuntimePermission ("*"));
    permissionsMap.put
        (DynamicPolicy.class.getProtectionDomain ()
         .getCodeSource (), permissions);
}
```

- The **main** method sets the policy in place, just as **MyPolicy** did; then it configures the policy through its own API, before calling the same **exercise** method:

```
public static void main (String[] args)
    throws Exception
{
    DynamicPolicy dynamic = new DynamicPolicy ();
    Policy.setPolicy (dynamic);
    System.setSecurityManager
        (new SecurityManager ());
    Permission p = new FilePermission
        ("dummy.txt", "delete");
    Permissions ps = new Permissions ();
    ps.add (p);
    dynamic.getPermissionsMap ().put
        (DynamicPolicy.class.getProtectionDomain ()
            .getCodeSource (), ps);
    exercise ();
}
```

- Run this class as an application and see the same results as with the completed **MyPolicy** demo.

```
build
runDynamic
Deleted dummy.txt.
```

- But note that from here, the class is highly configurable.
  - This can be a great technique when policies have to adapt to runtime conditions: for instance when JARs are added and removed from the code base either at runtime, or after build time and before runtime.

**Suggested time: 30-45 minutes**

In this lab you will enhance the NIH application by using the **DynamicPolicy** class. The current application requires that every new participant requires a separate **grant** in the policy file, which can quickly become a maintenance nightmare. You will use **DynamicPolicy** to grant necessary **FilePermissions** to participants on application startup, as they are discovered.

Detailed instructions are found at the end of the chapter.

# Privileged Actions and AccessControlContext

---

- **AccessController** studies the call stack on each call to **checkPermission**, and looks at policy grants of the requested permission on each distinct protection domain.
- The check succeeds only if all the domains have the required permission: this, again, is the **principle of least privilege**.
- Sometimes a piece of code wants to break that pattern, and for these scenarios **AccessController** offers various **doPrivileged** methods and a means of getting a snapshot of the current **AccessControlContext**.
- Many system classes exercise this option, so that they (on their own authority) can take full responsibility for specific actions they'll perform.
  - For instance the Serialization API uses a privileged action to read private fields from subject classes; without **doPrivileged**, the caller's own lower privileges would typically foil this attempt.
- We'll do more with privileged actions later, when we study JAAS.

## SUMMARY

- **We've seen the foundations of Java 2 security in this chapter.**
- **There is much more to consider:**
  - How to digitally sign Java code and set policies based on code signatures as well as code locations
  - How to design Java classes to take best advantage of language features and the security model itself
  - How to set digital signatures for application data, verify those signatures, and manage keys and certificates
  - How to grant permissions to users, and not just code bases – this is the Java Authentication and Authorization Service, or JAAS
- **We'll see all of these things in upcoming chapters, but every one of them relies on the basic mechanisms we've studied here.**
- **Even J2EE application servers rely on security managers and policy files, even if they do (for mostly historical reasons) mix in their own security models as well.**