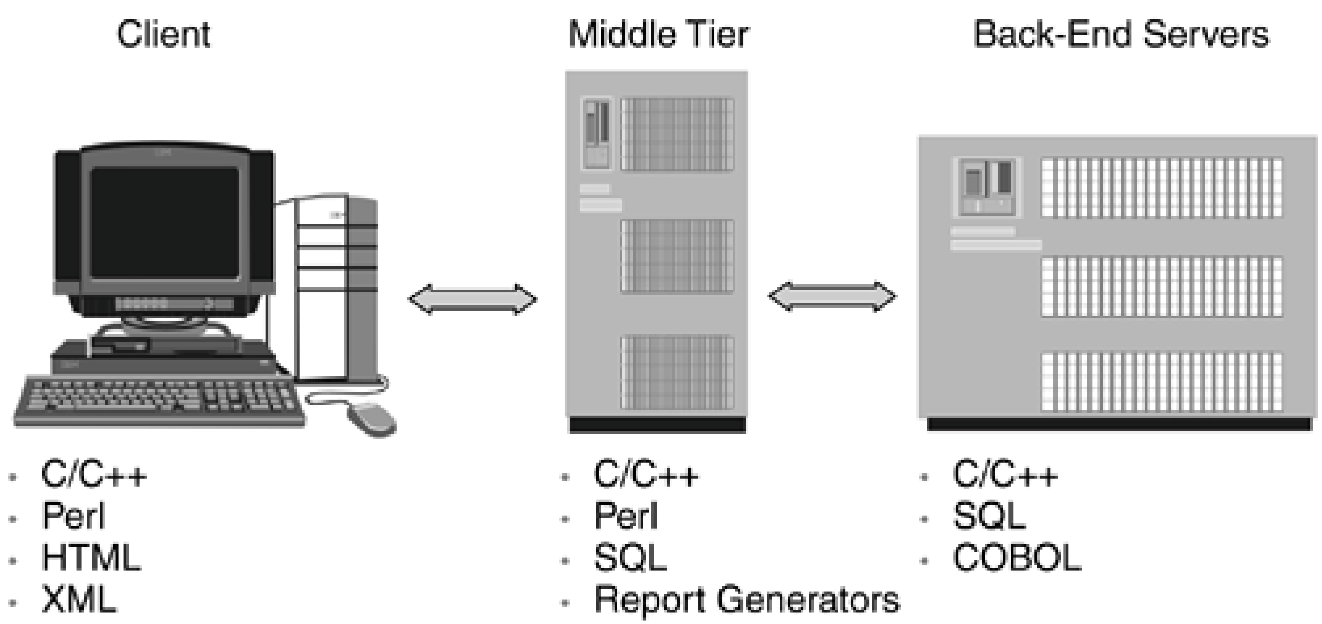
Java Security

Lecture 8

Java Technology

- Has been established as important for enterprise applications
  - To ease platform independent application development
    - Java Servlets, JavaServer Pages (JSP), Enterprise JavaBeans(EJB)
  - To provide security for e-business
    - J2EE builds on J2SE
      - Introduced fined-grained, policy-based security model that is customizable and configurable

Traditional Middle-tier Enterprise Environment

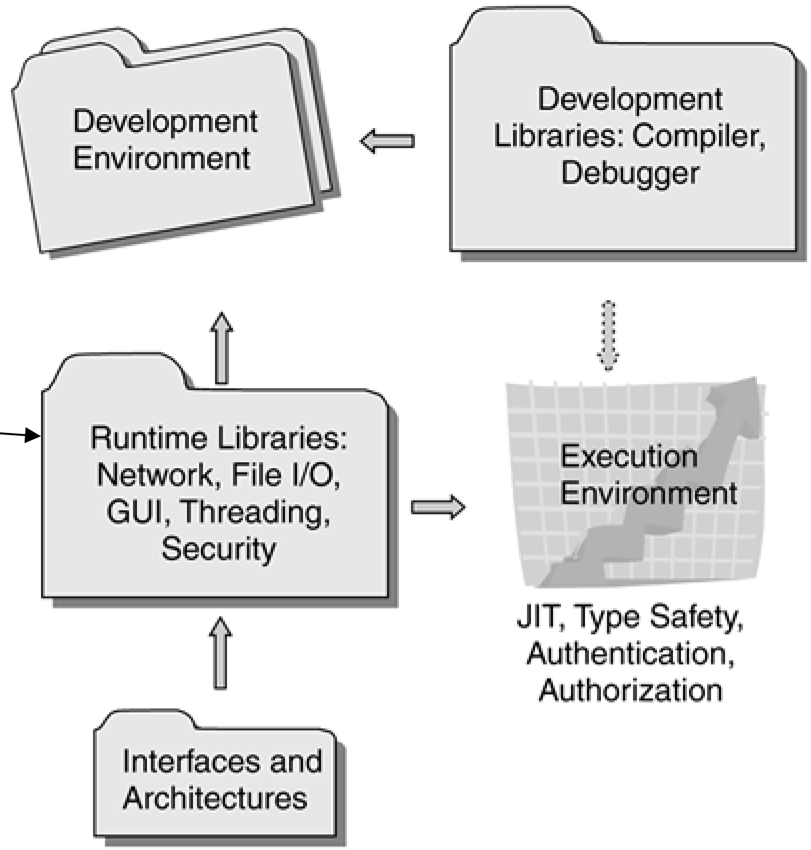


Java 2 Platform

- Programming language and runtime environment
  - In each tier
  - On multiple OSs
  - Libraries (WWW, Apache) such as for XML
- Additional frameworks are needed
  - To provide structure and design patterns that
    - Enable creating and deploying enterprise scalable applications.
- J2EE integrates Enterprise technologies
      - Interpreted through Java API
    - Distributed transaction support
    - Asynchronous messaging, and email
    - Portable Security technologies: Authentication, authorization, message integrity, and confidentiality
      - Enables interoperable security across the enterprise

Java Language Environment

- Java 2 SDK contain
  - Tools and library code for compilation and testing Java programs
- Libraries include
  - integrated support for various features
  - E.g., opening "socket" also includes defining proper authorization requirements
- Type-safety



Java Language Environment

- Execution Environment and Runtime
  - Mixed use of compiler and interpreter
  - Process compiled classes at execution time: JIT compilation
  - Provides security mechanisms
    - Type safety verification using dynamic type safety
      - E.g., array-bounds, type casting
    - When loaded into the JRE,
      - the code location is recorded,
      - If digitally signed, it is verified
        - For authorization
    - J2SE V1.4 also contains integrated authentication and authorization: JAAS Framework

Implemented as Java.Security.CodeSource

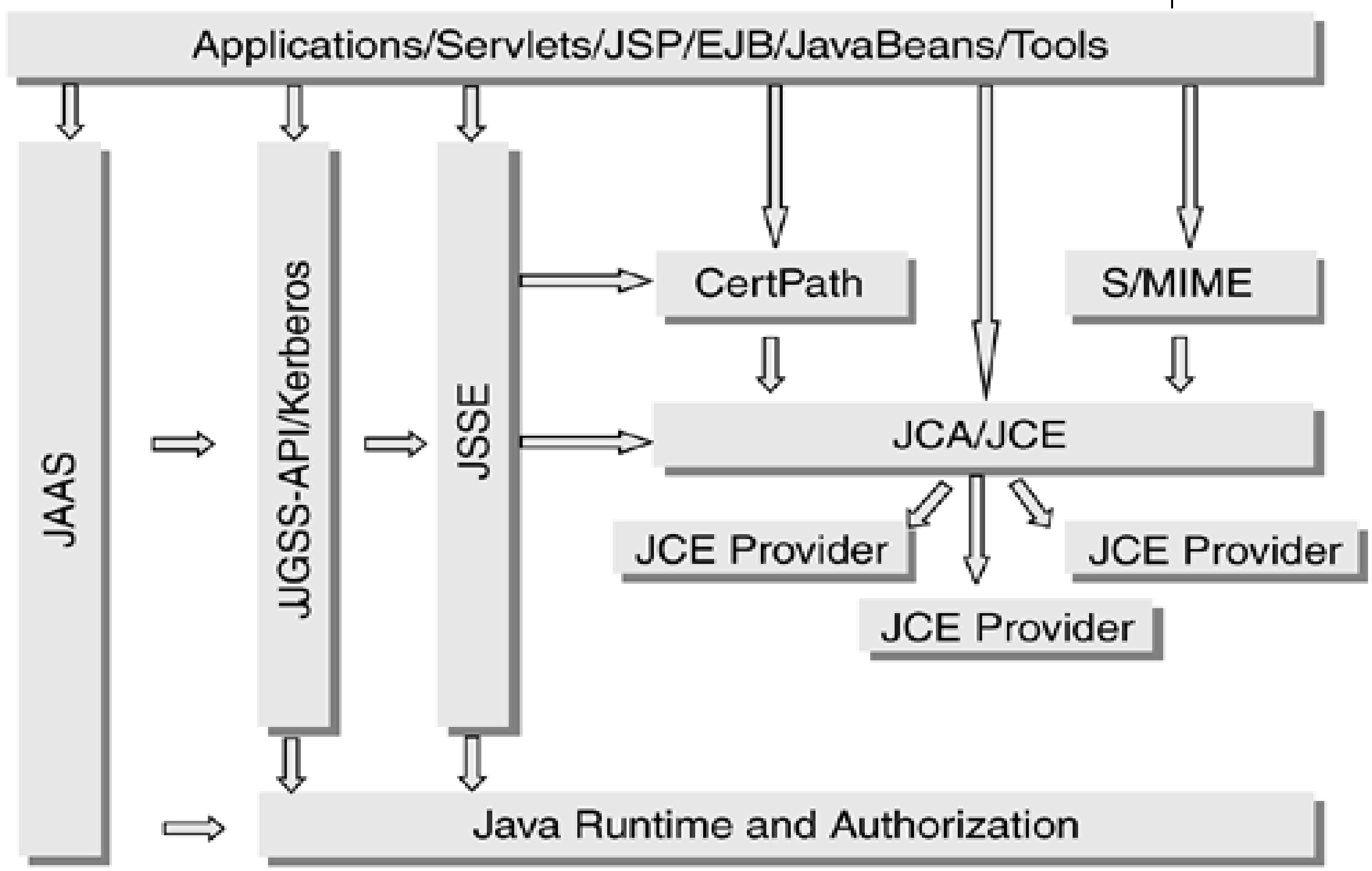
Java Language Environment

- Interface or APIs
  - Allows interaction with architected subsystems – where vendors provide services in a vendor netral manner
  - Allows interaction with external world
    - JDBC
    - JMS,
    - JCA,
    - JCE,
    - JAAS etc.

Java Security Technologies

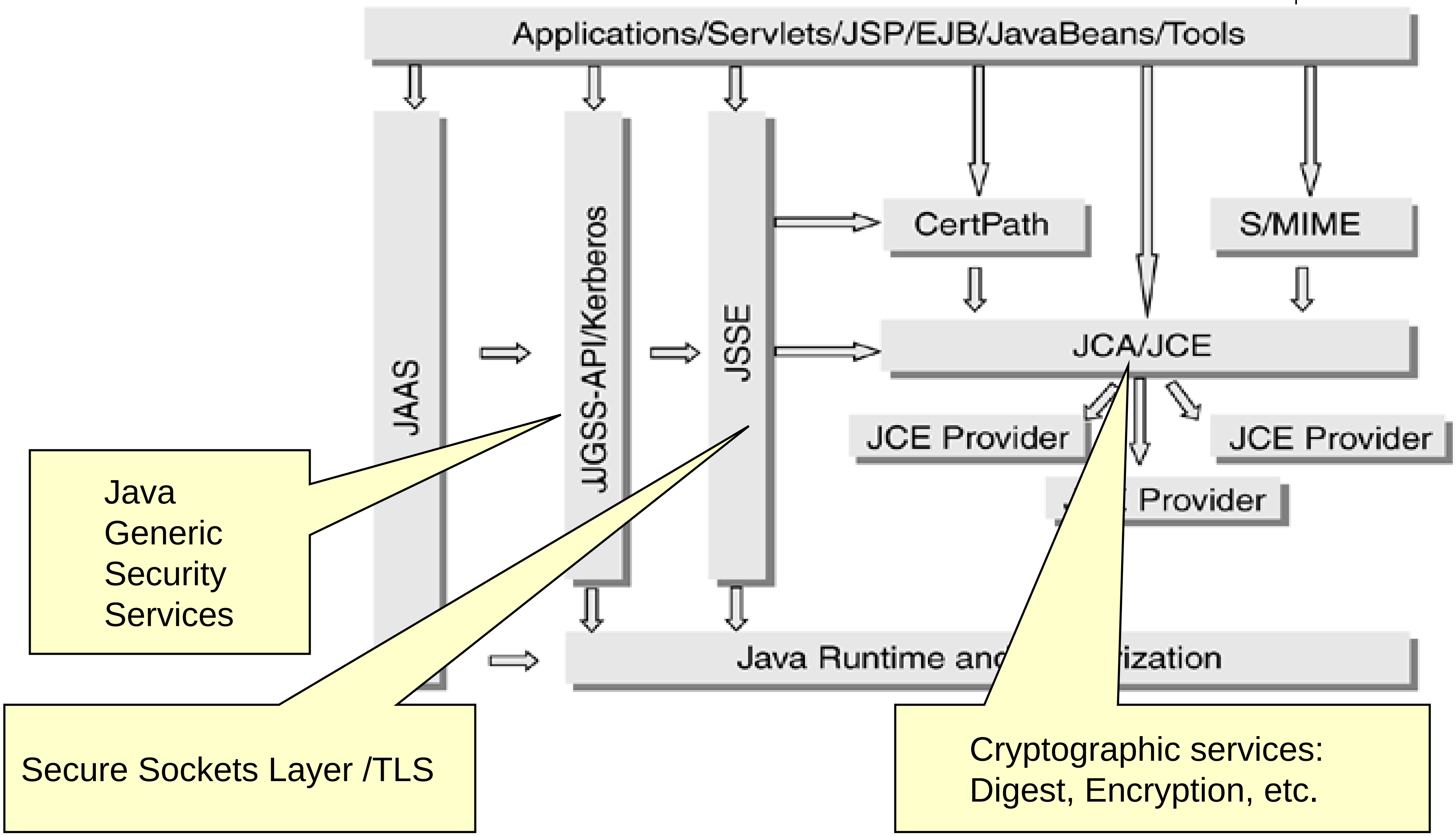
Integral,
Evolving, &
Interoperable

Security had been a primary Design goal



From Early days: Type Safety and Sandbox

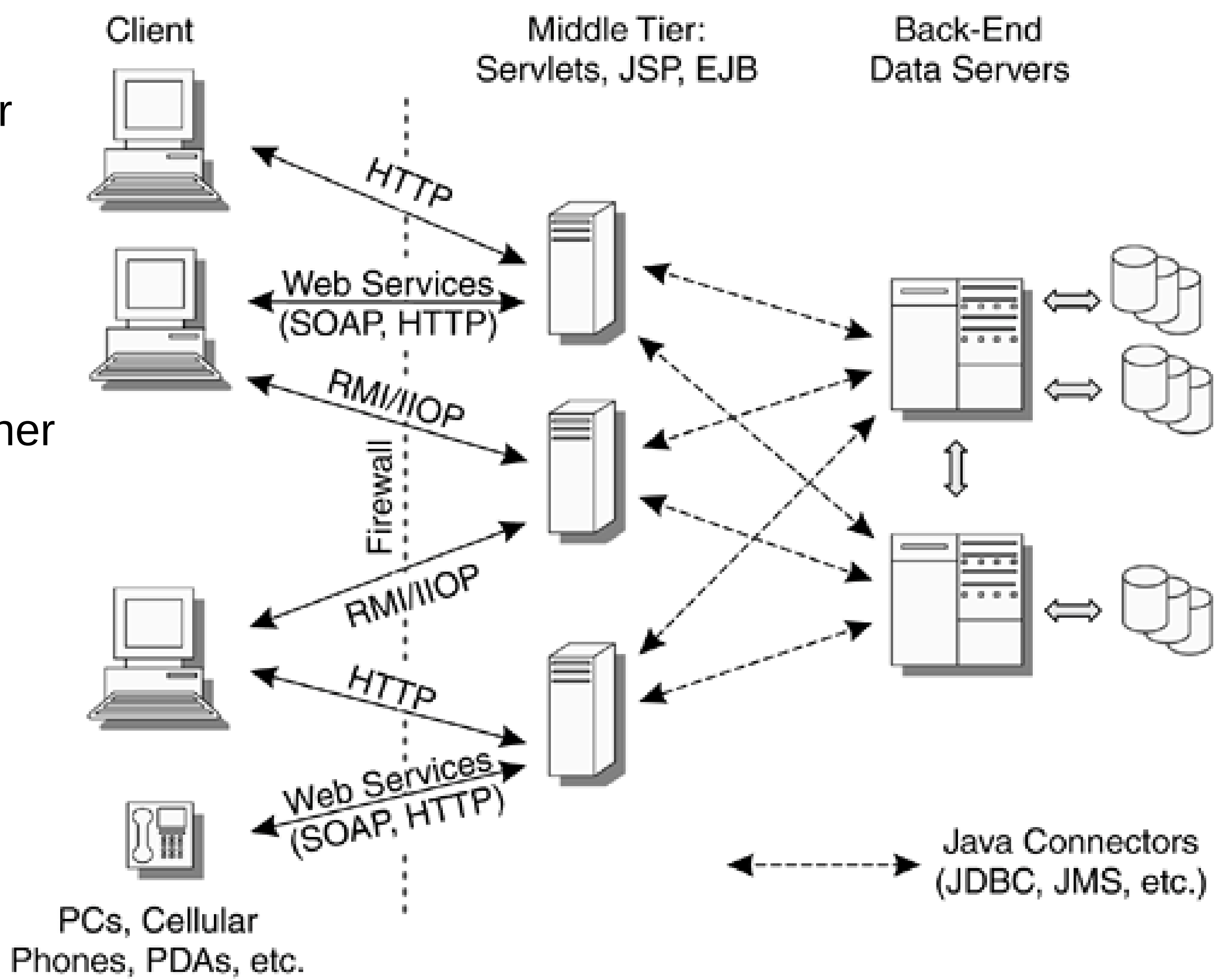
Java Security Technologies



Three tier model

Generalized into *N*-tier model

Java technology can be used in some tier and interfaced with other existing technology
- Java Connector Architecture (JCA)



Middle Tier

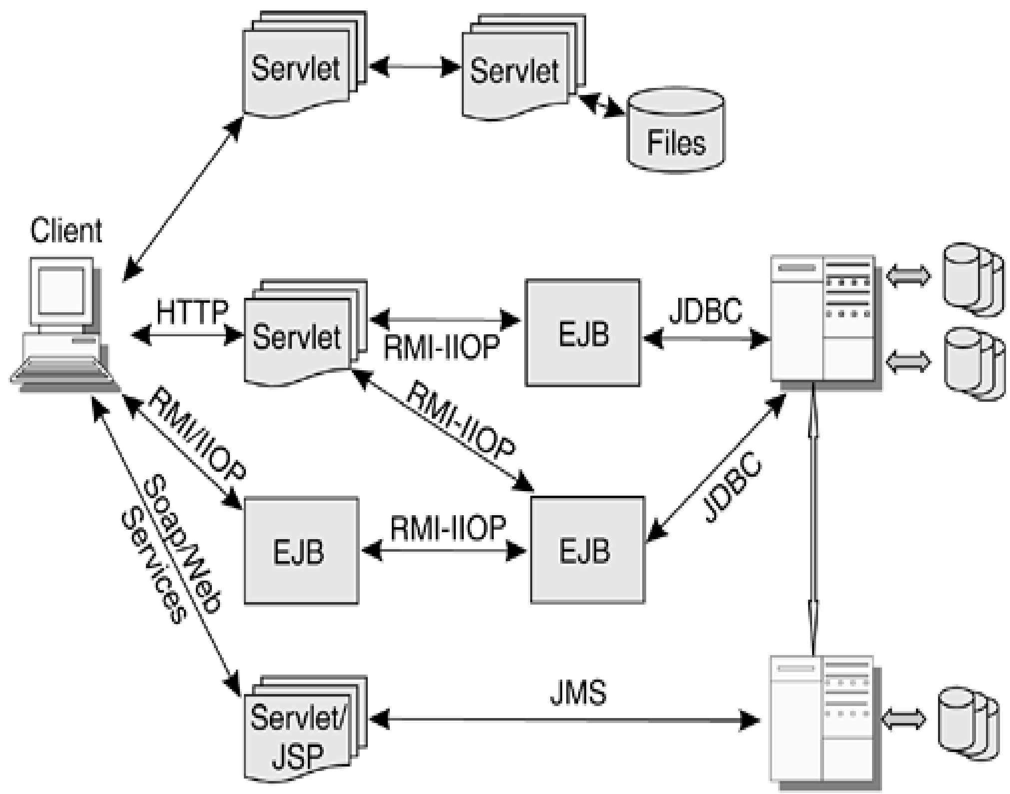
- CGI – original model for web servers
  - Did not scale well
    - Simple HTTP servers did not support multithreading
  - Lacked security
    - Buffer overflows, parameter validation issues, code injection, etc. were easier
- Java Servlet Programming model
  - Simplified server-side programming
  - Portable, and can use JCA to interface with others
  - Security services are part of the servlet architecture

Middle Tier

- Enterprise Java Beans
  - High throughput, scalability, and multiuser secure distributed transaction processing
    - Have constraints
      - Single threaded and may not read from file system
      - Need to use connectors to do I/O operations
    - Deployment descriptor (like in Servlets and JSP)
      - Include security requirements

Complex Application using J2EE

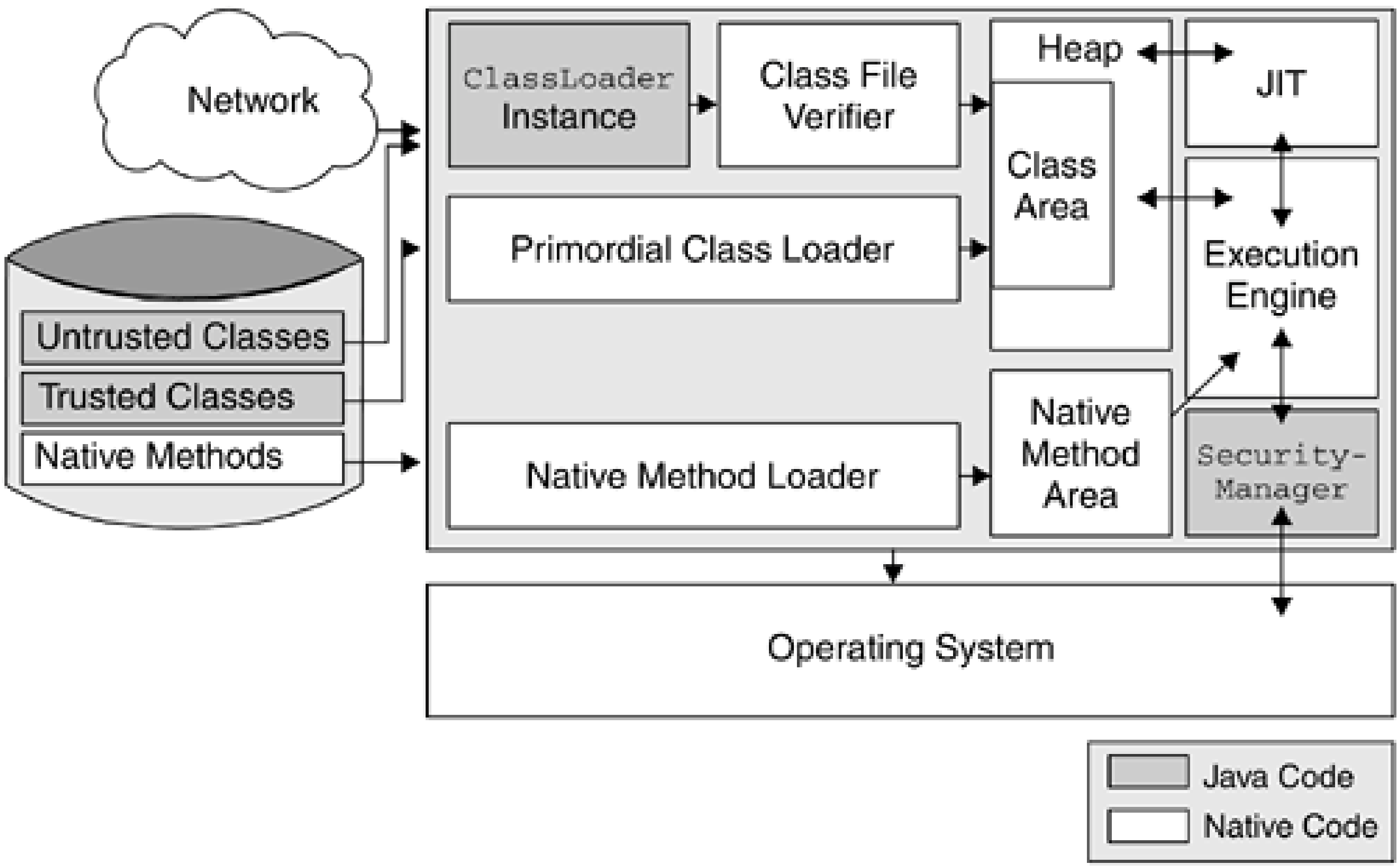
- Various protocols mediate communication between the client and server
  - HTTP,
  - Simple Object Access Protocol (SOAP)
  - Remote Method Invocation (RMI) over the Internet Inter-Object Request Broker (RMI-IIOP)
- Separation of components and their mediation by a container allows
  - Declarative policies



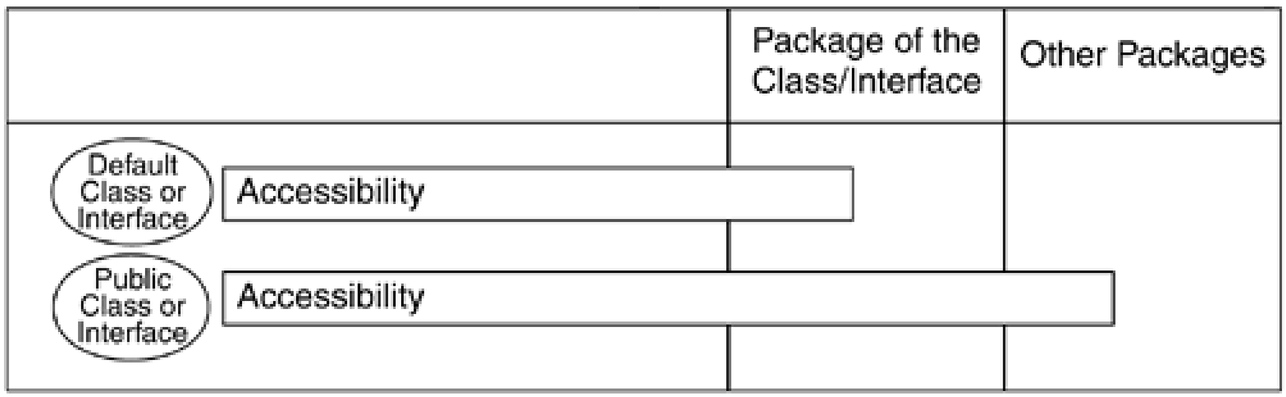
J2SE Security

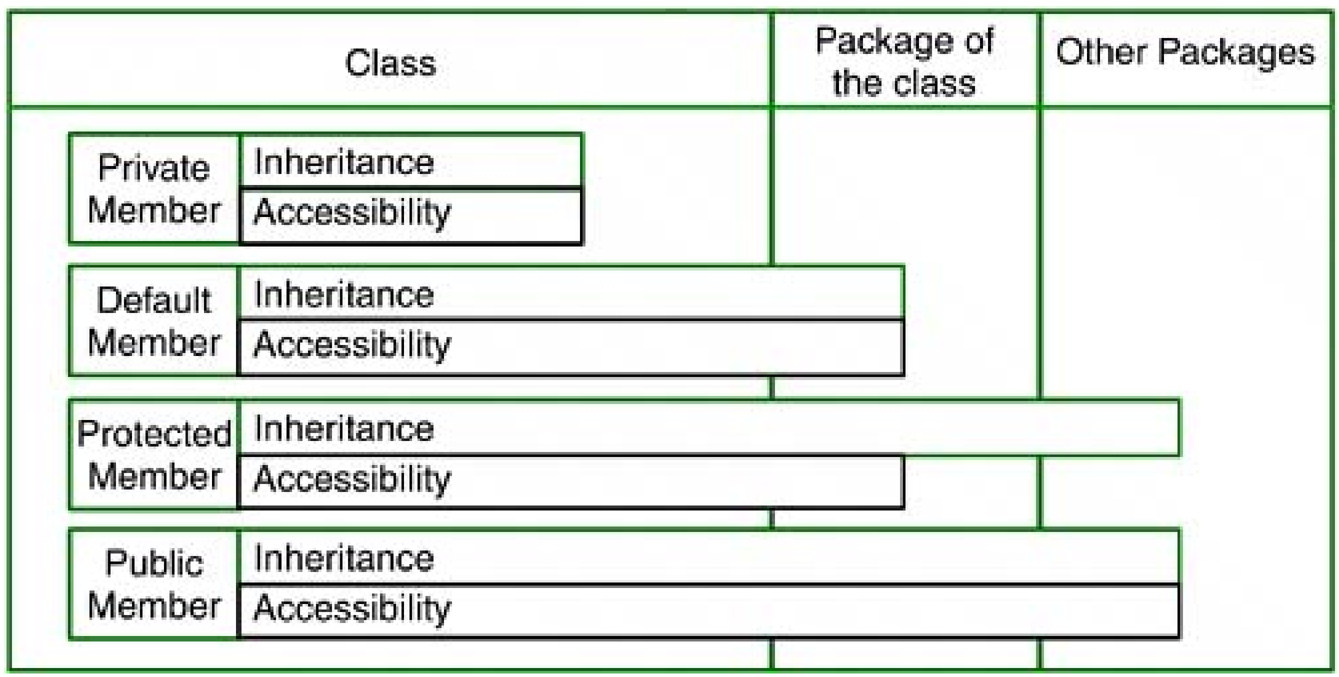
- Three legs of java security
  - Class loaders
    - Determine how and when to load code
    - Ensures that system-component within RE are not replaced with untrusted code
  - Class file verifier
    - Ensures proper formatting of nonsystem code
      - type safety requirements
      - Stacks cannot overflow/underflow
  - Security Manager
    - Enforces runtime access control restrictions on attempts to perform file and network I/O
    - Create a new class loader
    - Manipulate threads
    - Start processes
    - Terminate JVM
      - E.g., implements Java sandbox function

JVM components



Access to Classes, Interfaces, Fields, Methods





Class Loader

- Loading classes from a specific location
- Multiple class loaders may be active
- Set of classes loaded by a class loader is the class loader’s *name space*
- Security responsibilities
  - Name space separation
    - Avoid name clash problems
  - Package boundary protection
    - Can refuse to load untrusted classes into the core java packages, which contain the trusted system classes
  - Access-right assignment
    - Set of authorizations for each loaded class – uses security policy database
  - Search order enforcement
    - Establishes search order that prevents trusted classes from being replaced by classes from less trusted sources

Sources of code
- most trusted to least

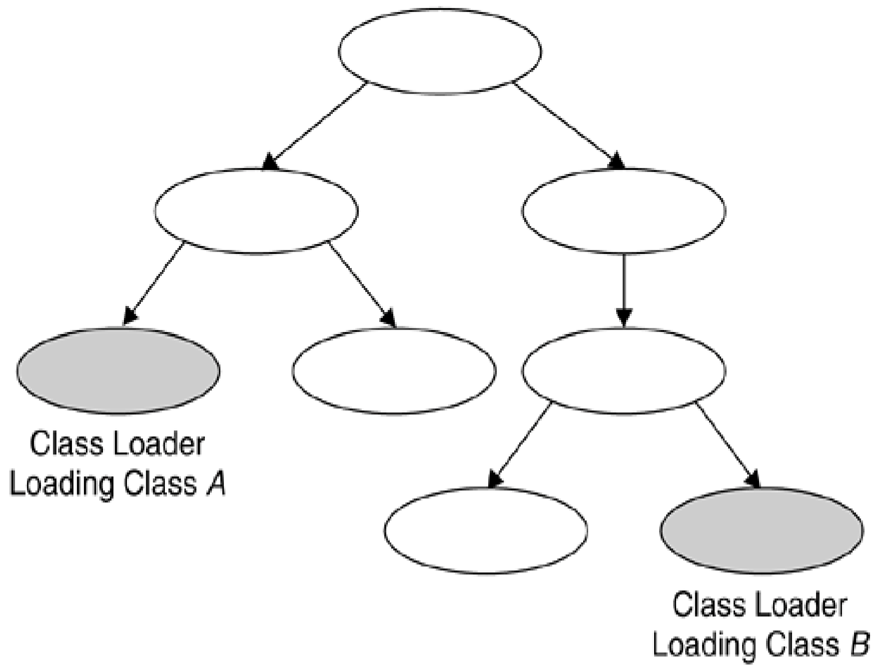
- Core classes shipped with JVM – system classes
  - E.g., java.lang, java.io, java.net
  - No restriction; no integrity verification
- Installed JVM extensions
  - E.g., Cryptographic service providers, XML parsers
- Classes from local file system
  - through CLASSPATH
- Classes from remote
  - Remote web servers

Class loader

- Must guarantee
  - Protection of trusted classes
    - When name classes occur, trusted local classes are loaded in preference to untrusted ones
  - Protection against name collision
    - Two classes with same name from different URLs
  - Protection of trusted packages
    - Otherwise, it could expose classes in trusted packages
  - Name-space isolation
    - Loading mechanism must ensure separate name-spaces for different class loaders
      - Classes from different name-spaces cannot interfere each other
    - Java class loaders are organized in a tree structure

Class loader

- A cannot directly
  - instantiate B,
  - invoke static methods on B or
  - instance methods on objects of type B
- Many class loaders may be active at any given time



Loading classes from Trusted Sources

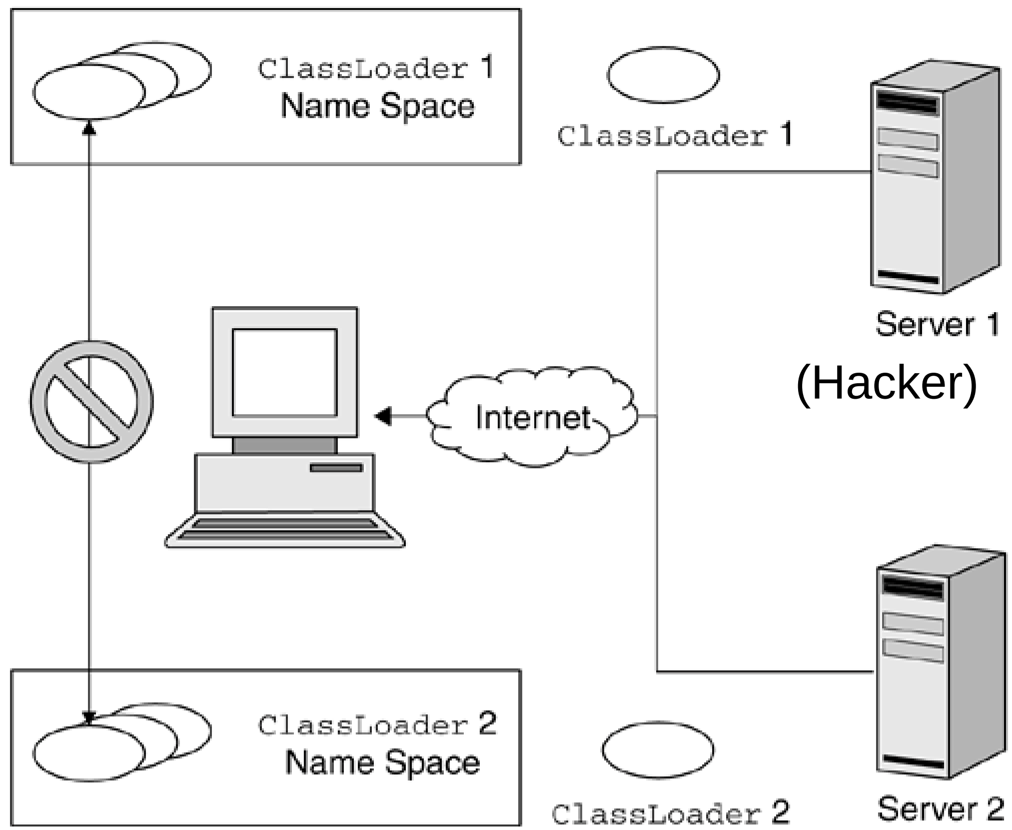
- Primordial class loader
  - Built in JVM; also known as internal, or null, or default class loader
  - Loads trusted classes of java runtime
  - Loaded classes are not subject to verification
  - Not subjected to security policy restriction
    - These are located using boot class path (in Java 2)

Loading classes from untrusted Sources

- Sources include
  - Application classes, extension classes and remote network locations
- Application class loader
  - Users classes; not trusted; not by primordial
  - URLClassLoader an implementation of the java.lang.ClassLoader
  - Application class path from CLASSPATH
  - Uses URLs to locate and load user classes
  - Associate permissions based on security configuration
- Extension classes
  - Trust level is between Application and fully trusted system classes
  - Typically granted all permissions
  - Added to extension class path – should be allowed to trusted users only

Loading classes from untrusted Sources

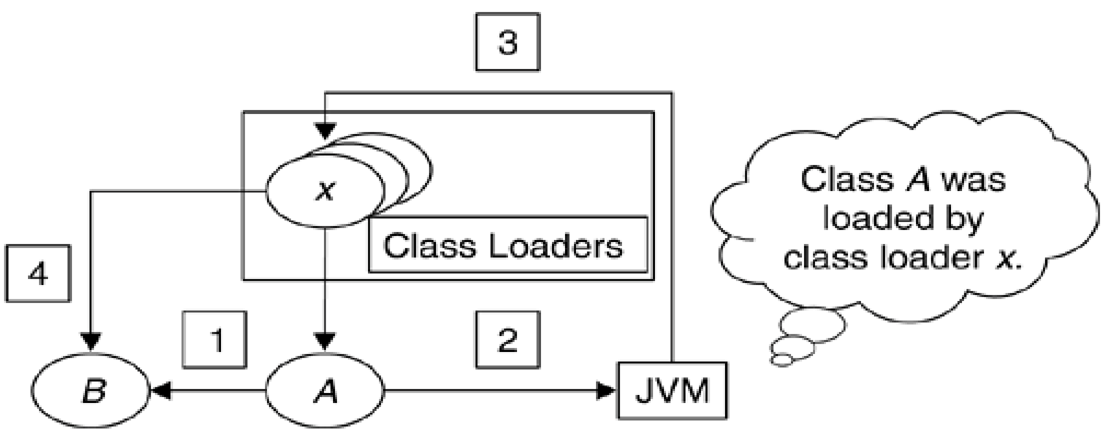
- Classes from Remote Network – least trusted
  - A class loader is created for each set of URLs
  - Classes from different URLs may result in multiple ClassLoaders being created to maintain separate name spaces
  - Safety and integrity verification checks
  - Run confined in sandbox



(Bank)

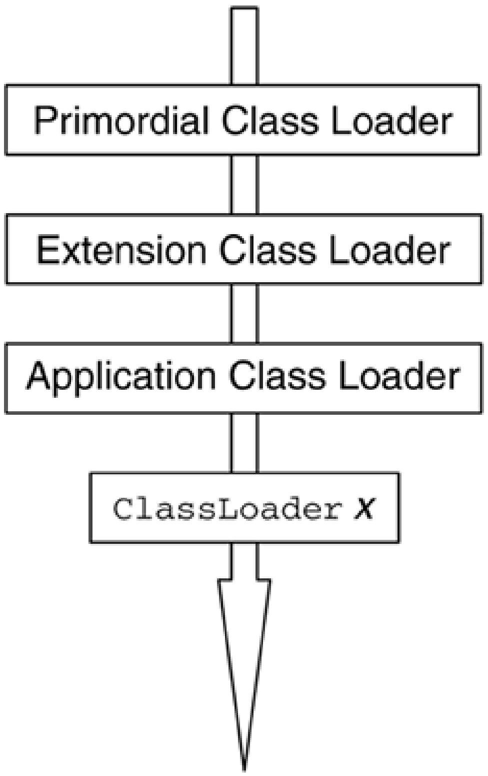
Enforcing order - Design

- Class A is loaded by x
- A references B; hence class loader needs to load B
  - If x was primordial, getClassLoader() = null
- If B already loaded
  - Checks A has permissions (x interacts with SecurityManager)
  - Returns reference to object
- Else loader checks with SecurityManager to see if A can create B
  - If yes, checks the boot class path first -> extension class path -> application class path -> network URL in that order
  - If found in other than boot class path, verification is done



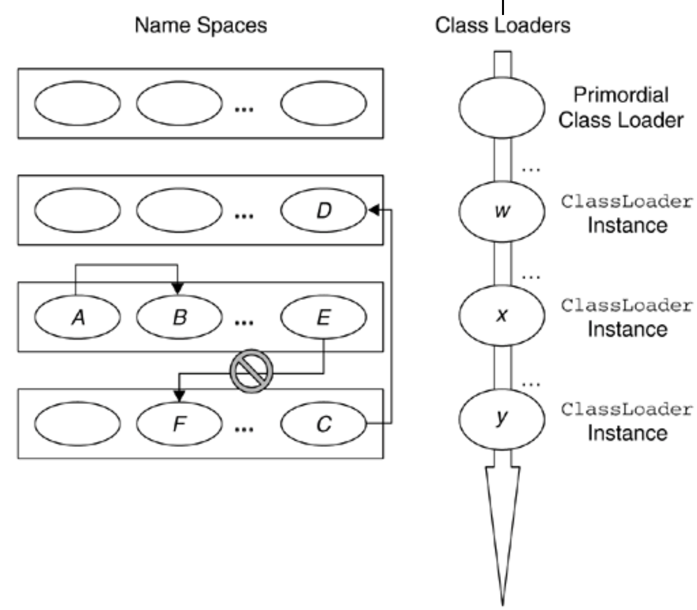
Delegation hierarchy - Implementation

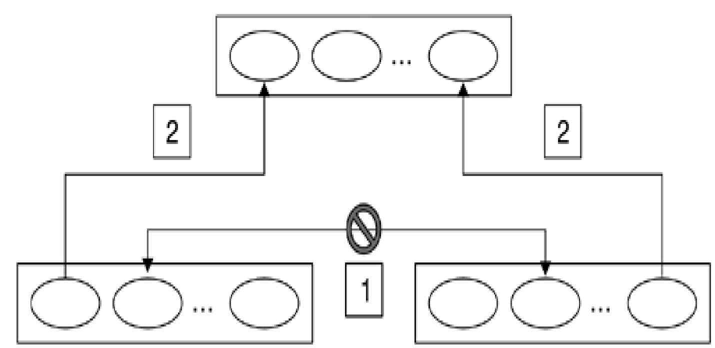
- Primordial class loader
  - In general is not a java class
  - is generated at JVM startup (not loaded)
- Every ClassLoader class needs to be loaded
  - When a program instantiates a ClassLoader, the program's class loader becomes the ClassLoader's parent
    - E.g., extention class loader is created at JVM start-up by one of the JVM's system programs, whose class loader is the primordial class loader – hence primordial class loader is parent
  - Forms parent/child relationships



Referencing classes

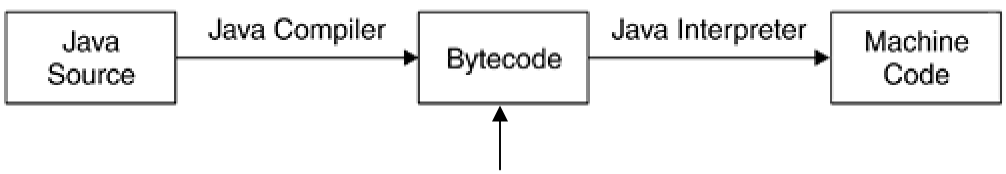
- The delegation model guarantees
  - A more trusted class cannot be replaced by the less trusted
  - A and its instance can call B and its instances if both were loaded by the same class loader
  - C and its instance can call D and its instances if D’s class loader is an ancestor of C’s loader
  - E and its instance cannot call F and its instances E’s class loader is an ancestor of C’s loader
  - Classes in name space, created by different class loaders cannot reference each other
    - Prevents cross visibility
    - How can such classes exchange information?





Class Verifier

- At this point following is guaranteed
  - Class file loaded
    - Cannot supplant core classes
    - Cannot Inveigle into trusted packages
    - Cannot interfere with safe packages already loaded
  - However the class file itself may be unsafe
- Key sources of unsafe byte code
  - Malicious java compiler
    - byte code may itself be from non-Java programs
  - Class editors, decompilers, disaasemlers



Can be easily edited by hex class editor

ByteCode Example

```
 0: CA FE BA BE 00 00 00 2E 00 ID 0A 00 06 00 0F 09 Eb9<...........
10: 00 10 00 11 08 00 12 0A 00 13 00 14 07 00 15 07 ................
20: 00 16 01 00 06 3C 69 6E 69 74 3E 01 00 03 28 29 .....<init>...( )
30: 56 01 00 04 43 6F 64 65 01 00 0F 4C 69 6E 65 4E V...Code...LineH
40: 75 6D 62 65 72 54 61 62 6C 65 01 00 04 6D 61 69 umberTable...mai
50: 6E 01 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 6E 67 n...([Ljava/lang
60: 2F 53 74 72 69 6E 67 3B 29 56 01 00 0A 53 6F 75 /String;)V...Sou
70: 72 63 65 46 69 6C 65 01 00 0F 48 65 6C 6C 6F 57 rceFile...HelloW
80: 6F 72 6C 64 2E 6A 61 76 61 0C 00 07 00 08 07 00 orld.Java.......
90: 17 0C 00 18 00 19 01 00 0B 48 65 6C 6C 6F 20 57 .........Hello W
                              1B 00 1C 01 00 0A 48 orld..........H
                              01 00 10 6A 61 76 61 elloWorld...Java
                              65 63 74 01 00 10 6A /lang/Object...j
                              53 79 73 74 65 6D 01 ava/lang/System.
                              6A 61 76 61 2F 69 6F ..out...Ljava/io
                              65 61 6D 3B 01 00 13 /PrintStream;...
100: 6A 61 76 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 java/io/PrintStr
110: 65 61 6D 01 00 07 70 72 69 6E 74 6C 6E 01 00 15 eam...println...
120: 28 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 (Ljava/lang/Stri
130: 6E 67 3B 29 56 00 20 00 05 00 06 00 00 00 00 00 ng;)V. .........
140: 02 00 00 00 07 00 08 00 01 00 09 00 00 00 1D 00 ................
150: 01 00 01 00 00 00 05 2A B7 00 01 Bl 00 00 00 01 .......*•..±....
160: 00 0A 00 00 00 06 00 01 00 00 00 01 00 09 00 0B ................
170: 00 0C 00 01 00 09 00 00 00 25 00 02 00 01 00 00 .........%......
180: 00 09 B2 00 02 12 03 B6 00 04 Bl 00 00 00 01 00 ..².....¶..±.....
190: 0A 00 00 00 0A 00 02 00 00 00 05 00 08 00 06 00 ................
1A0: 01 00 0D 00 00 00 02 00 0E .........
```

```
class HelloWorld
{ public static void main(String args[])
        { System.out.println("Hello World");
        }
}
```

Class Verifier

- Bytecode can be easily modified to change the behavior of the class using such hex editors
- Decompilers can recreate source code
  - It can then be modified to create malicious byte code using a custom compiler
  - Disassembler generates pseudo assembly code, which can be modified and reassembled back to corrupted java code

Class Verifier

- Class editors, decompilers and dissemblers can also be used to perpetrate privacy and intellectual property attacks
  - Valuable algorithm can be broken
  - Security mechanism can be revealed and bypassed
  - Hard-coded confidential information (keys, password) can be extracted
- A break in release-to-release compatibility can cause a class to be unsafe
  - A member that was accessible is not available
  - A member has changed from static to instance
  - New version has different return, number and type parameters
- All these need to be checked by Class Verifier !

Duties of Class File Verifier

- Some possible compromise to the integrity of JVM as follows
  - Forge illegal pointers
    - Class confusion attack: obtain reference to an object of one type and use it as another type
  - Contain illegal bytecode instructions
  - Contain illegal parameters for bytecode instructions
  - Overflow or underflow the program stack
    - Underflow – attempting to pop more values than it pushed
    - Overflow – placing values on it that it did not remove
  - Perform illegal casting operation
  - Attempt to access classes, fields or methods illegally

Check the size of stack before and after each call

Tag each object with type

Class Verifier

- Four passes based on Sun JVM
  - Over the newly loaded class
  - Any pass fails the class is rejected
  - First three before the execution and the last during the execution
- Pass 1: File-integrity check
  - Checks for a signature
    - The first four bytes is magic number 0xCAFEBABE
  - Check that the class itself is neither too long nor too short – otherwise throws exceptions

Class Verifier

- Pass 2: Class-integrity check – ensures
  - Class has a superclass unless it is Object
  - Superclass is not a final class
  - Class does not override a final method in its superclass
  - Constant pool entries are well formed
  - All the method and field references have legal names and signatures

Class Verifier

- Pass 3: bytecode-integrity check – the bytecode verifier runs
  - Checks how the code will behave at runtime
    - Dataflow analysis, static type checking
- Bytecode verifer is responsible for ensuring
  - Bytecodes have correct operands and their types
  - Data types are not accessed illegally
  - Stack is not overflowed/underflowed
  - Method calls have appropriate parameters

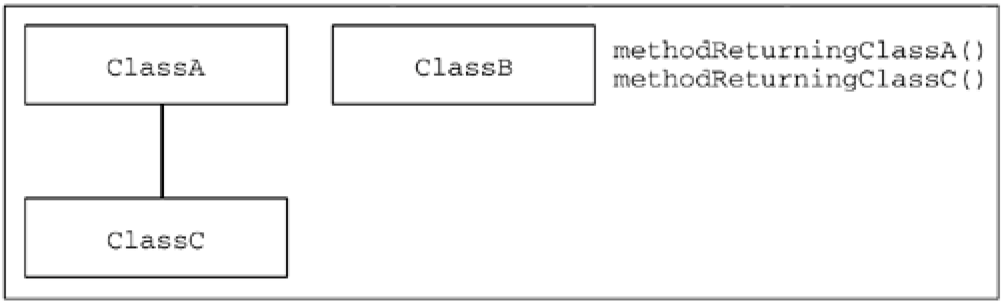
Class Verifier

- The result indicates a class file in one category
  - Runtime behavior is demonstrably safe (accept)
  - Runtime behavior is demonstrably unsafe (reject)
  - Runtime behavior is neither demonstrably safe nor demonstrably unsafe
    - Cannot be completely eliminated
    - Means bytecode verifier is not enough to prevent runtime errors – some runtime checking is required

Class Verifier

- Pass 4: Runtime-integrity check
  - Bytecode verification cannot confirm certain behavior

```
ClassB b = new ClassB();
ClassA a = b.methodReturningClassA();
```




```
ClassB b = new ClassB();
ClassA a = b.methodReturningClassC();
```

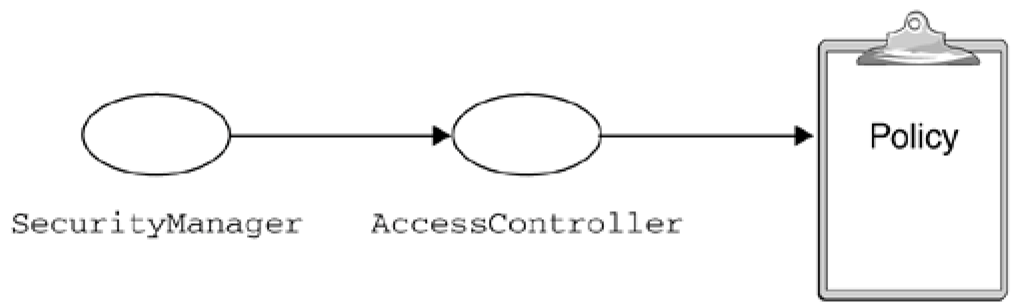
Class files are loaded only when a method call is executed or a field in an object of that class is modified

Security Manager

- Java environment attacks can be
  - System modification
    - A program gets read/write access
  - Privacy invasion
    - Read access to restricted information
  - Denial of service
    - Program uses up system resources without being invited
  - Impersonation
    - Masquerades as a real user of the system
- Security manager enforces restriction against first two attacks and to some extend the last

Security Manager

- SecurityManager – concrete class
  - Implementation supports policy driven security model
  - Resource-level, access control facility
  - checkPermission(Permission object)



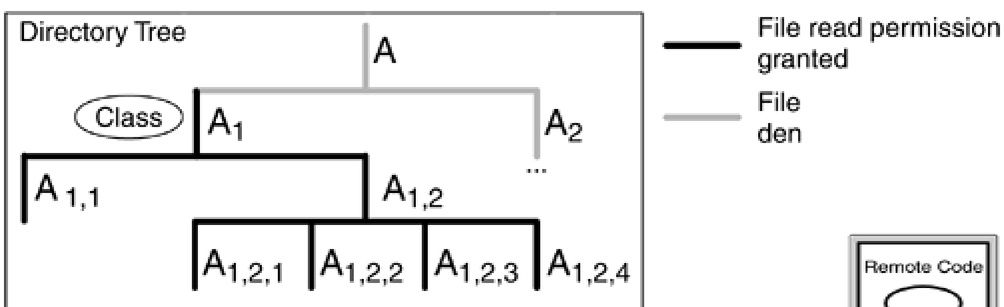
Security Manager

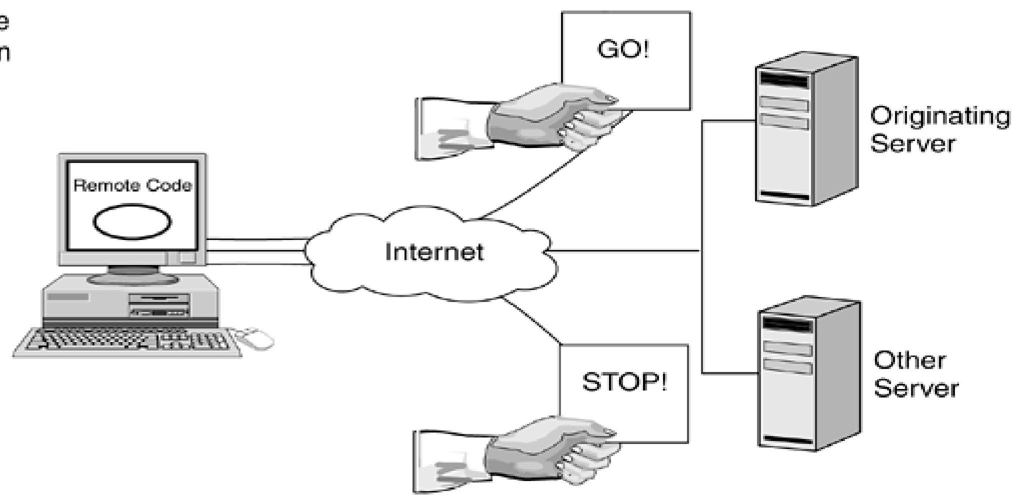
- Resources protected by default SecurityManager

Areas of Control	Method Names	Permission Types Passed to checkPermission()
Network	checkAccept()	SocketPermission
	checkConnect()	SocketPermission
	checkListen()	SocketPermission
	checkMulticast()	SocketPermission
	checkSetFactory()	RuntimePermission
Thread	checkAccess()	RuntimePermission
File System	checkDelete()	FilePermission
	checkRead()	RuntimePermission, FilePermission
	checkWrite()	RuntimePermission, FilePermission
Operating System	checkExec()	FilePermission
	checkPrintJobAccess()	RuntimePermission
	checkSystemClipboardAccess()	AWTPermission
	checkLink()	RuntimePermission
	checkTopLevelWindow()	AWTPermission
JVM	checkExit()	RuntimePermission
	checkPropertyAccess()	PropertyPermission
	checkPropertiesAccess()	PropertyPermission
	checkAwtEventQueueAccess()	AWTPermission
	checkCreateClassLoader()	RuntimePermission
Packages and Classes	checkPackageAccess()	RuntimePermission
	checkPackageDefinition()	RuntimePermission
	checkMemberAccess()	RuntimePermission
Security	checkSecurityAccess()	SecurityPermission

Security Manager

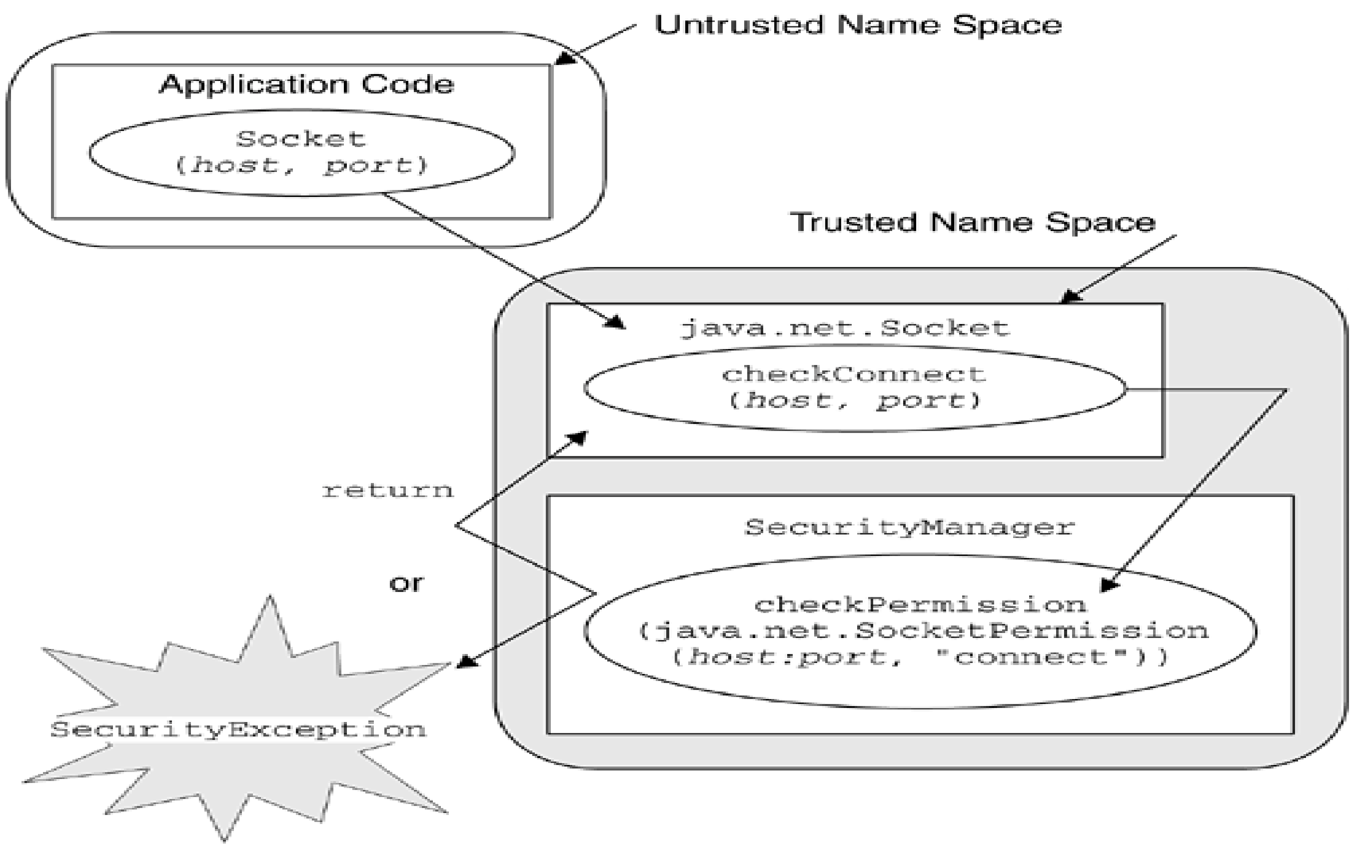
- SM Automatically grants
  - a class file java.io.FilePermission necessary to read to all files in its directory and subdirectory
  - Java.net.SocketPermission that allows remote code to connect to, accept, and resolve local host and the host the code is loaded from





Security Manager Operation

- Once installed, a SecurityManager is active only on request – it does not check anything unless one of its check methods is called by other system functions



Types of attacks

- Some of the security holes in previous java releases
  - Infiltrating local classes
    - JVM implementation bug: allowed an applet to load a class from any directory on the browser system
      - OS should be configured to restrict writing access to the directories pointed to by the boot class path
      - Extension framework are by default granted full access to the system resources – only trusted users should be allowed to add extensions to the runtime environment

Types of attacks

- Type confusion
  - If an attacker can create an object reference that is not of the type it claims to be, there is possibility of breaking down protection. JVM flaws
    - JVM access checking that allowed a method or an object defined as private in one class to be accessed by another class as public
    - JVM bug that failed to distinguish between two classes with the same name but loaded by different class loaders

Types of attacks

- Network lookpholes
  - Failure to check the source IP address rigorously
    - This was exploited by abusing the DNS to fool SM in allowing the remote program to connect to a host that would normally have been invisible to the server
- JavaScript backdoors
  - Exploit allowed script to persist after the web page has been exited
- Malicious code: Balancing Permission
  - Cycle stealing
  - Impersonation

Interdependence of three legs

- Although have unique functions, they are inter-dependent
  - Class-loading mechanism relies on SM to prevent trusted code from loading its own class loader
  - SM relies on class-loading mechanism to keep untrusted classes and local classes separate name spaces and to prevent the local trusted classes from being overwritten
  - Both the SM and CL system rely on class file verifier to make sure that class confusion is avoided and that class protection directives are honored.
- If an attacker can breach one of the defenses – the security of the whole system can be compromised

Java 2 Permission Model

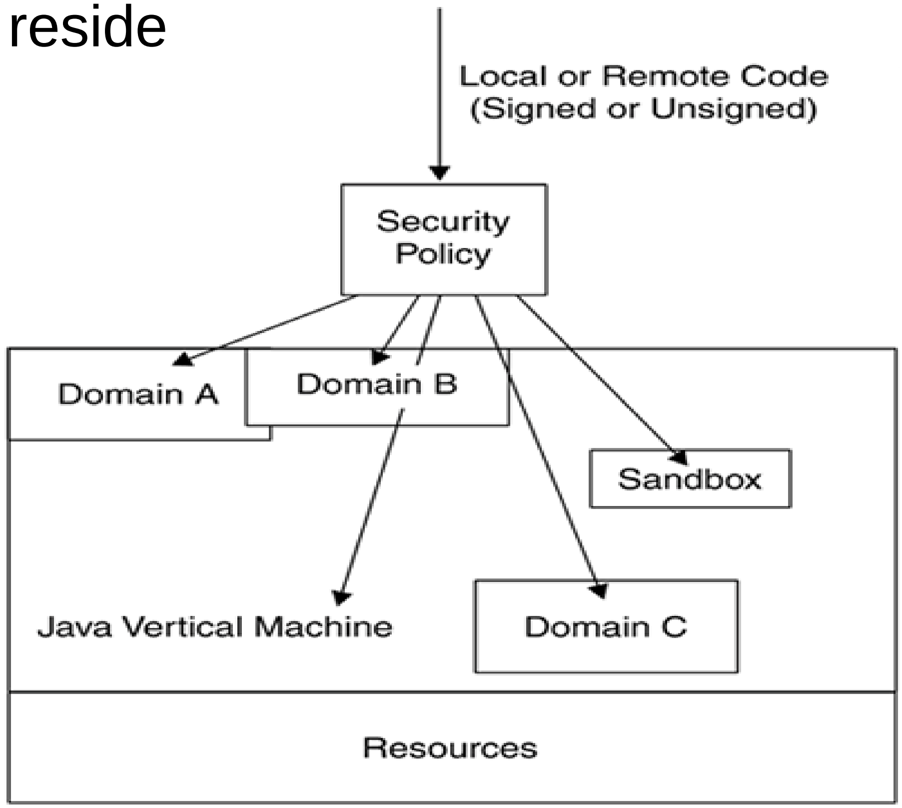
- fine-grained access control model
  - Ability to grant specific permissions to a particular piece of code about accessing specific resources
    - Based on the signers of the code, and
    - The URL location from which code was loaded
  - System admin can specify permission on a case-by-case basis
    - the policy database is by default implemented as a flat file, called *policy profile*

Java 2 Permission Model

- In multiuser system, a default system policy data bases can be defined, and each user can have a separate policy database
- Additionally in an intranet, network admin can define a corporate wide policy database and install it on a policy server for all the Java systems in the network to download and use
  - At runtime, (corporate wide policy database + system policy database + user-defined policy database) gives the current security policy in effect

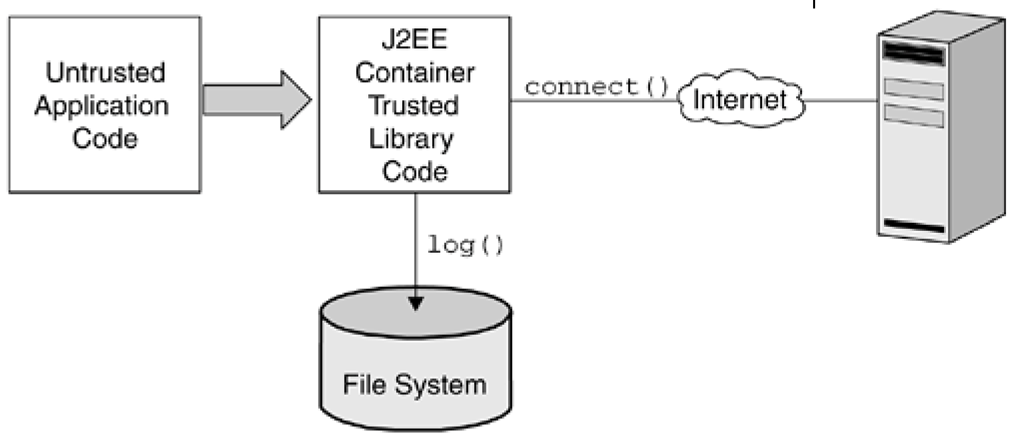
Java 2 Access control mechanism

- Predetermined security policy of the java system dictates the Java security domains within which a specific piece of code can reside



Lexical scoping of privilege modifications

- A piece of code can be defined as privileged



Trusted code called opens socket connection and logs to a file all the times it has been accessed

Caller should have java.net.SocketPermission but not necessary to have java.io.FilePermission

```
someMethod()
{
   // unprivileged code here...
   AccessController.doPrivileged(new PrivilegedAction()
   {
      public Object run()
      {
         // privileged code goes here, for example:
         System.loadLibrary("awt");
         return null; // nothing to return
      }
   });
   // unprivileged code here...
}
```

Java 2 Security Tools

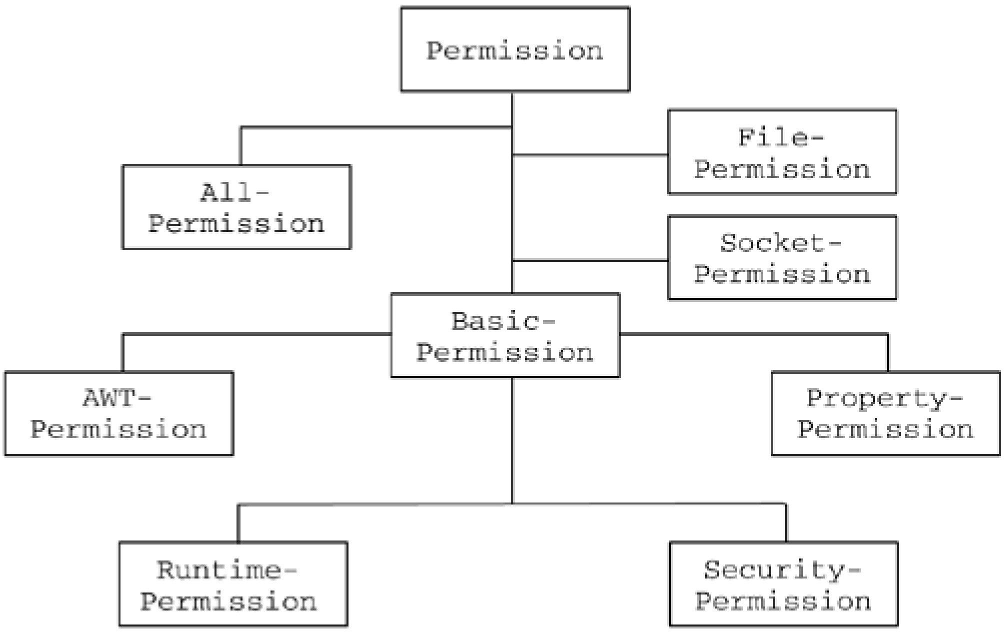
- jar utility
  - Aggregates and compresses collections of java programs and related resources
  - Only JAR files can be signed/sealed
- keytool utility
  - Creates key pairs; imports/exports X.509 certificates; manages keystore
  - Keystore – protected database containing keys/certificates
- jarsigner utility
  - To sign JAR files and to verify signatures of JAR files
- Policytool
  - To create and modify policy configuration files

Java Authentication and Authorization Service

- Basic java security model
  - Grants permissions based on code signers and URL locations
    - Insufficient in enterprise environment – as concept of user running the code is not captured
- JAAS complemented basic model by taking into account users running the code

Java Permissions

- java.security package contains abstract Permission class
  - Subclasses define specific permission



Permissions API inheritance tree

Specific permission class generally in packages in which they are most likely to be used, e.g.,
  FilePermission in java.io package
  SocketPermission in java.net package

Java Permissions

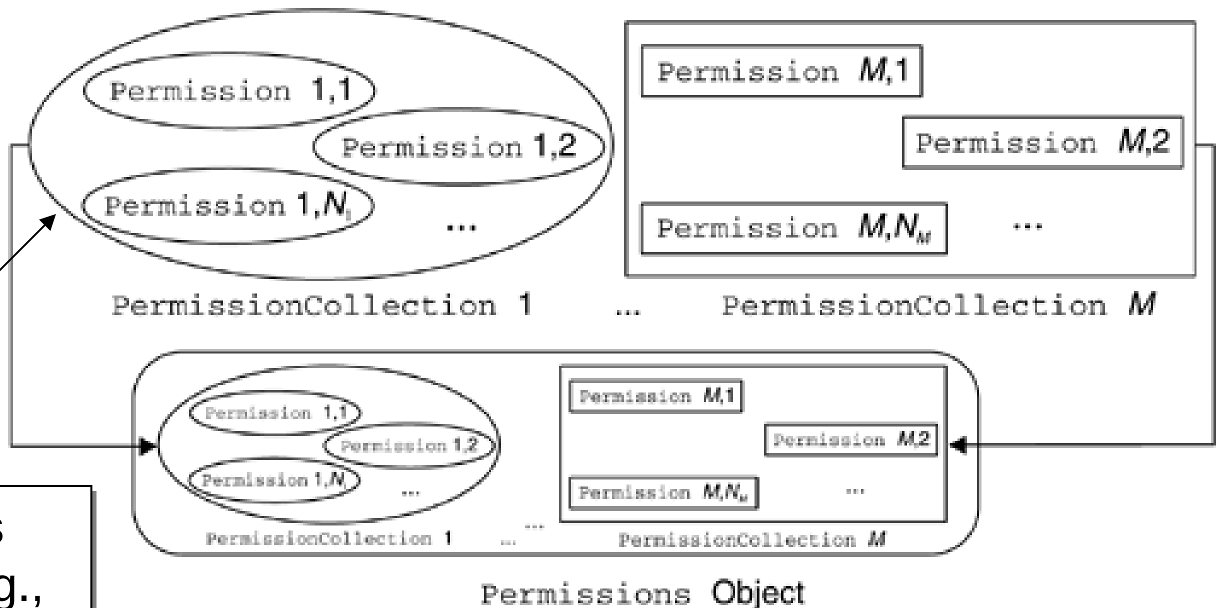
- Permission may have
  - A target and optional actions (access mode)
  - E.g., both target and action included
    - java.io.FilePermission “C:\AUTOEXEC.BAT”, “read, write, execute”
  - E.g., target only
    - java.io.RuntimePermission “exitVM”
  - E.g., no target
    - java.security.AllPermission – full access to all system resources

Java Permissions

- Classes
  - PermissionCollections and Permissions

Abstract class

Final class



Homogeneous permission; e.g., file permissions

Group of heterogeneous permission objects

Permission class

- implies() method – abstract method that returns true
  - a implies b means
    - Granting an application permission a autmatically grants it permission b also.
      - Giving AllPermisions implies granting rest of the permissions
      - java.io.FilePermission “/tmp/*”, “read” implies java.io.FilePermission “/tmp/readme.txt”, ‘read”

AllPermissions

- Care should be taken when granting AllPermissions and any of the following Permissions
  - Permission to define the systems SecurityManager; E.g.,
    - RuntimePermissions “createSecurityManager” and RuntimePermissions “setSecurityManager”
  - Permission to create a class loader
    - Delegation hierarchy may not be respected
  - Permission to create native code
    - Native code runs on OS and hence bypasses java security restrictions
  - Permission to set the system’s security policy

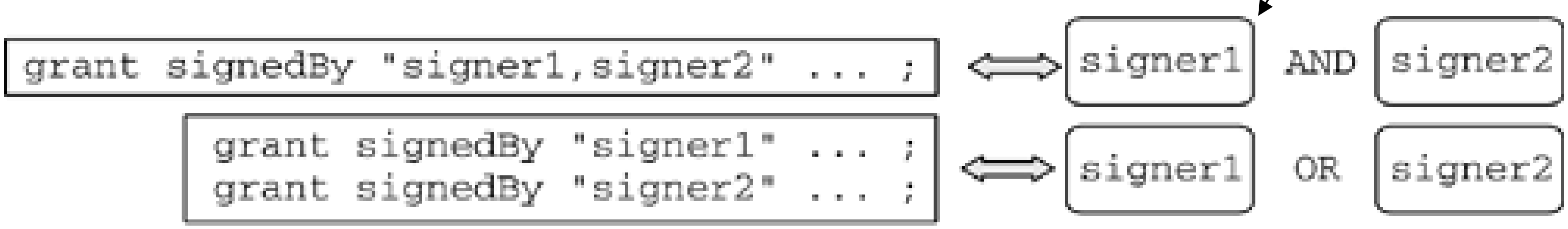
Java Security Policy

- Policy can be configured – declarative
  - Can also be easily changed
  - java.security.policy can be subclassed to develop customized policy implementation (e.g., encrypted file instead of flat files)

```
grant [signedBy signers][, codeBase URL] {
permission Perm_class [target][, action][, signedBy signers];
[permission ...]
};      //GRANT Entry syntax
```

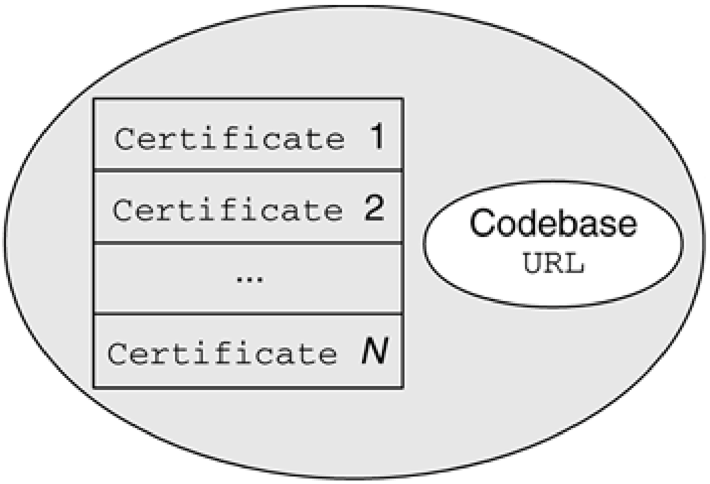
```
grant signedBy "bob, alice" codeBase "http://www.ibm.com" {
permission java.io.FilePermission "C:\AUTOEXEC.BAT", "read";
permission java.lang.RuntimePermission "setSecurityManager";
};// GRANT entery
```

Keystore used by JVM should have certificates of bob AND alice. To do OR, duplicate the grant statement



Multiple policy files
Code source

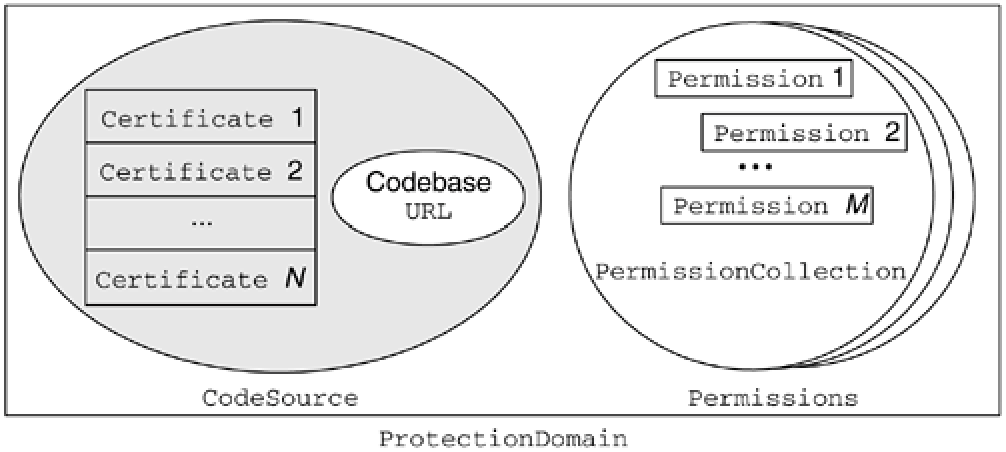
- Can be combined at runtime to form single policy object
  - No risk of conflict as only positive permissions
  - By default program is denied any access
- CodeSource
  - Codebase is the URL location that the code is coming from
  - If two classes have the same codebase and are signed by the same signers – they have the same CodeSource



CodeSource

Protection domain

- When a class is loaded into JVM
  - CodeSource of that class is mapped to the Permissions granted to it by the current policies
  - Class loader stores CodeSource and Permissions object into a ProtectionDomain object
    - That is -> Based on the class's CodeSource the ClassLoader builds the ProtectionDomain for each class

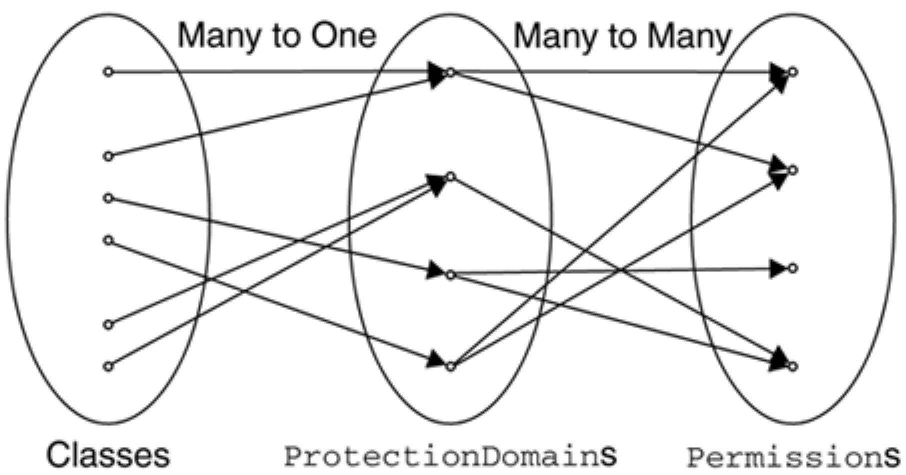


System and Application domains

- System classes are fully treated
  - ProtectionDomain (system domain) is pre-built that grants AllPermissions (also known as null protection domain)
- Application domain
  - Non system classes
  - Zero or more application domains
    - As many application domains as there are non-system CodeSource

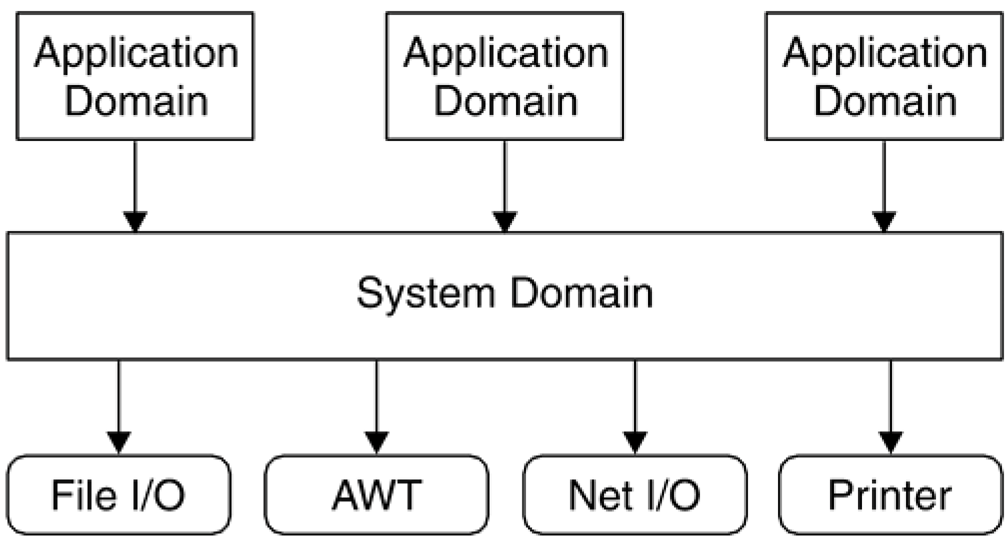
Relationships

- All the classes with the same CodeSource belong to the same ProtectionDomain
- Each class belongs to one and only one ProtectionDomain
- Classes that have the same Permissions but are different from CodeSources belong to different ProtectionDomains



Basic Java 2 Access Control Model

- SecurityManager.checkPermission() is called to allow access to resources
  - It is an interface
  - Actually relies on AccessController.checkPermission() to verify the permission has been granted



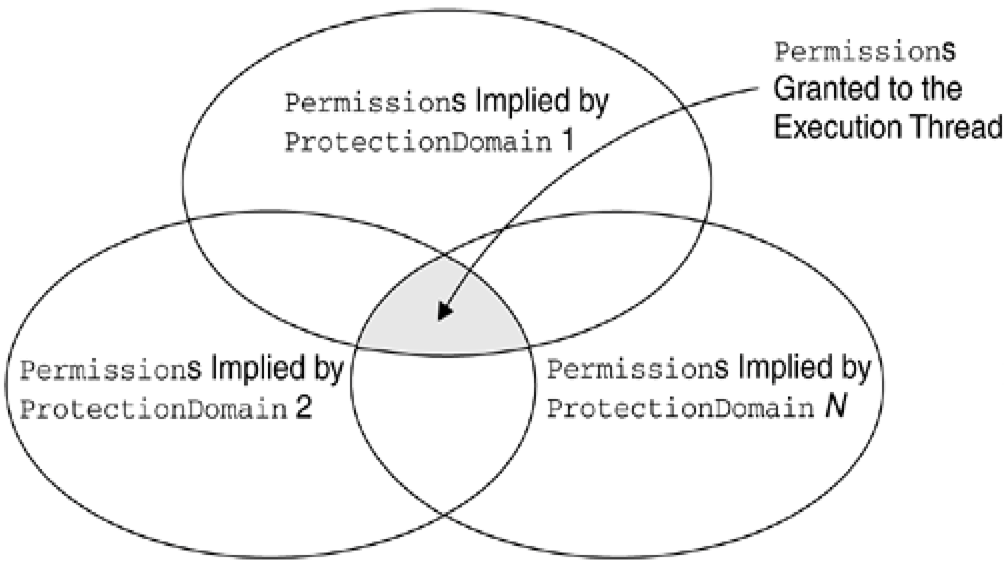
Basic Java 2 Access Control Model

- Thread of execution
  - may occur
    - Completely within a single Protection domain (e.g., the system domain), or
    - May involve one or more application domains and also the system domain
  - Contains a number of stack frames – one for each method invocation
    - Each stack frame is mapped to the class in which the method is declared

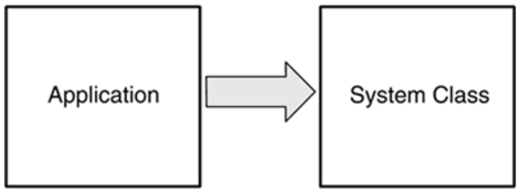
Basic Java 2 Access Control Model

- AccessController.checkPermission()
  - Walks through each thread's stack frames, getting the protection domain for each class on the thread's stack
  - As each ProtectionDomain is located, the implies() method is invoked to check if Permission is implied by the ProtectionDomain
    - Repeats until the end of the stack is reached
    - If all the classes in the frame have the Permission to perform the operation – the check is positive
    - If even one ProtectionDomain fails to imply the permission – it is negative

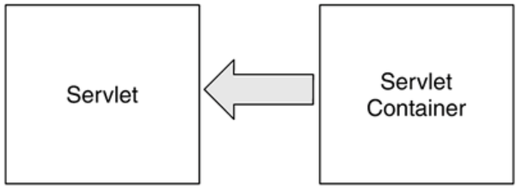
Basic Java 2 Access Control Model



Examples



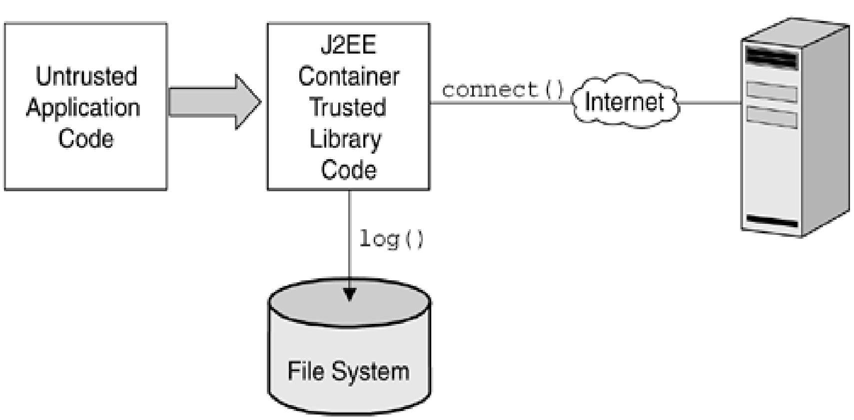
Less privileged to more privileged

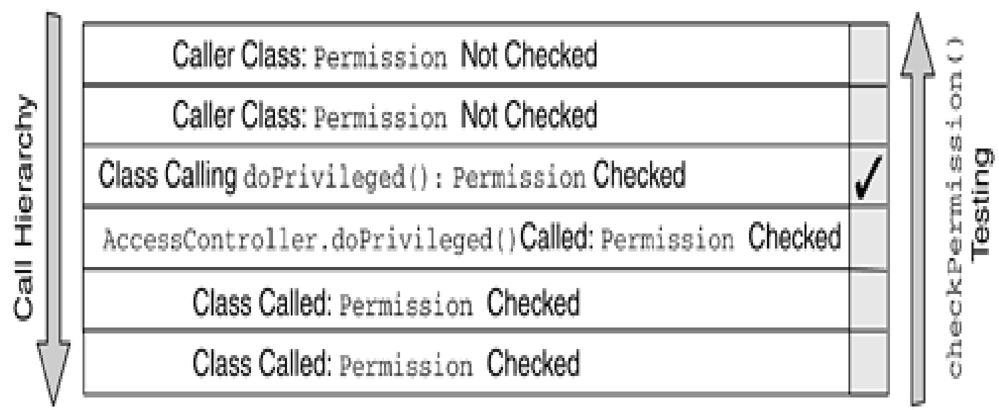


More privileged to less privileged

Privileged Code

- Intersection of permission of the ProtectionDomain can be a limitation
  - Controlled solution: Wrap the needed code into
    - AccessController.doPrivileged() to see whether Permission being checked is implied
      - The search stops at the stack that implies





Privileged Code