Software Security

Case Study:

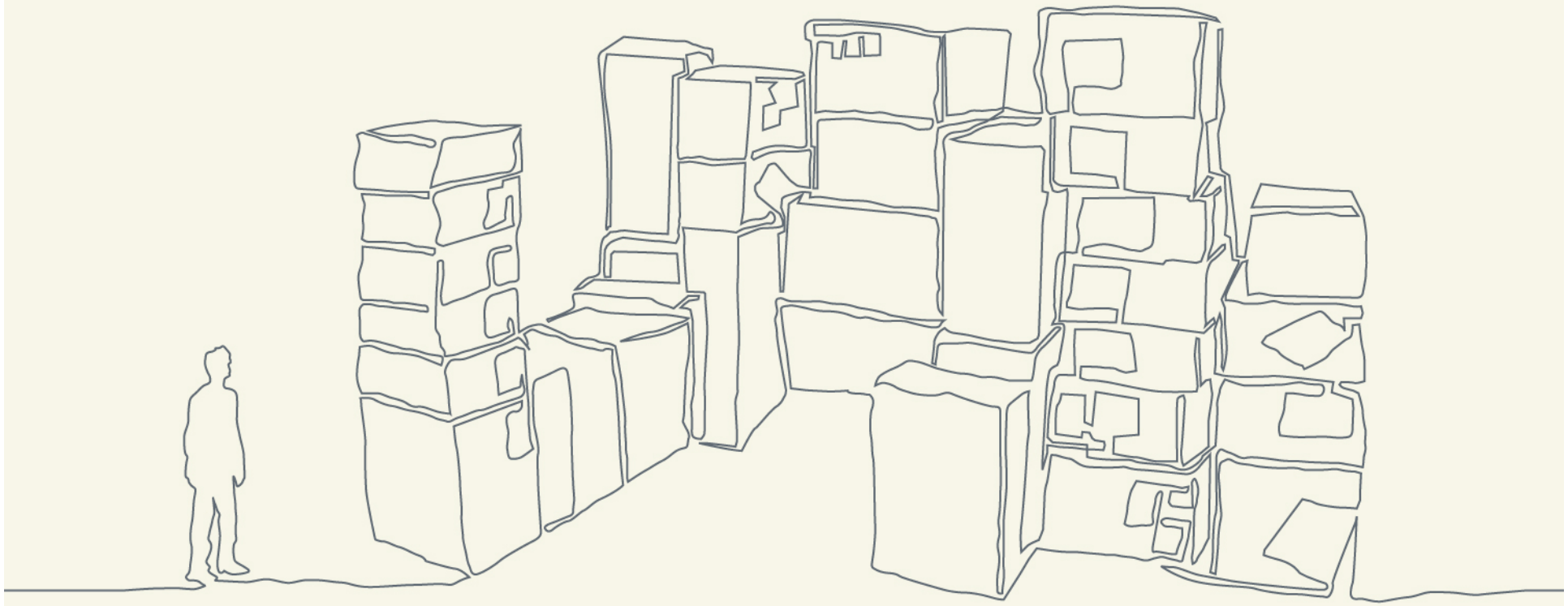
Java 2 Security

heathwallace

Agenda

- About Us
- Objectives
- Introduction
- Foundations of Java 2 Security
- Tools and APIs
- Conclusion

About Us



About HeathWallace

- Dominant supplier of online services to the FS Industry Globally
- Extensive experience with High Net Worth segment
- Right mix of skills and experience to deliver a robust, differentiated online experience
- Currently working on projects in Europe, North America, South America, Asia and the Middle East

Key Facts

- 60 Internet professionals in UK and Hong Kong
    - Journey planning, Research, IA, Design and Build
- Global strategic online agency for HSBC
    - 2006, 2 billion site visits to HSBC web sites
    - 27% of all credit cards now opened online
- Global agency for RBS and GE Money


RBS
The Royal Bank of Scotland


HSBC


ABN·AMRO


AIB


Standard
Chartered


euronext


BARCLAYS


mashreq


STANDARD LIFE


charles SCHWAB


UBS


ASHBURTON


AXA
INVESTMENT
MANAGERS


NatWest


Schroders


GE Money


Coutts


ZURICH


Lombard

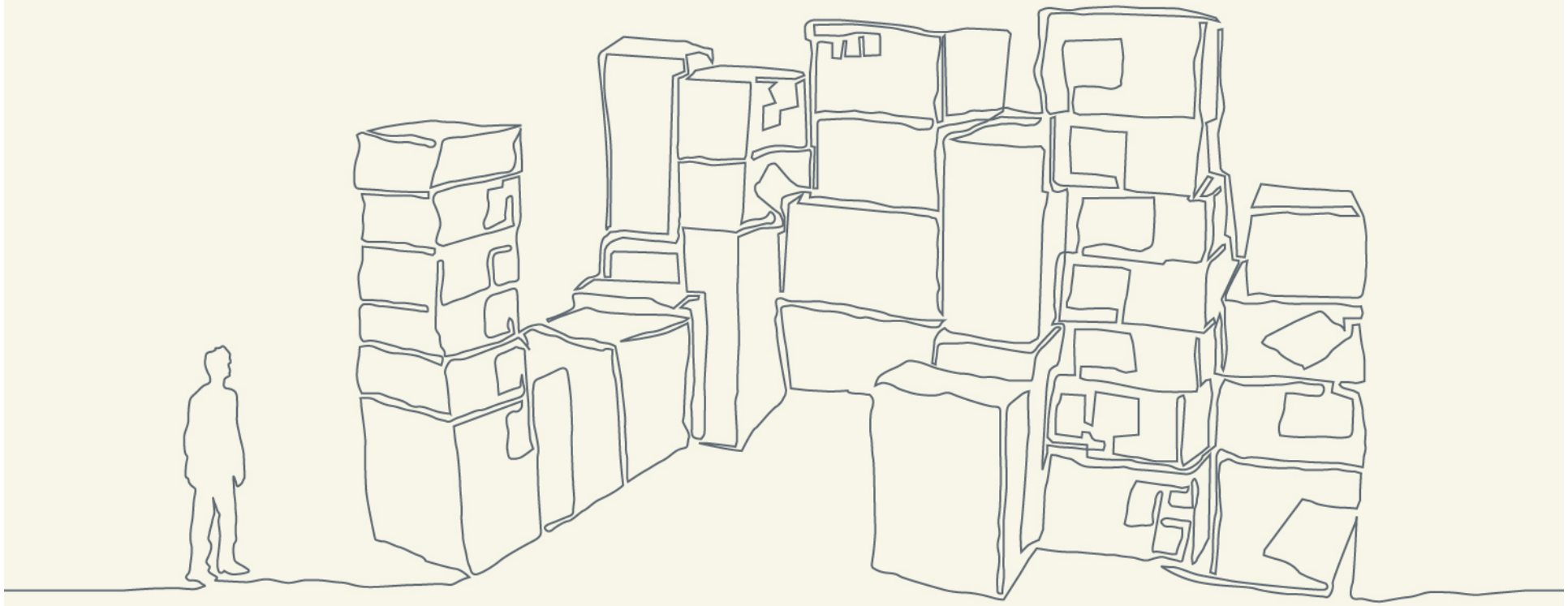

marbles™


GAM


Barclaycard


HFC
BANK

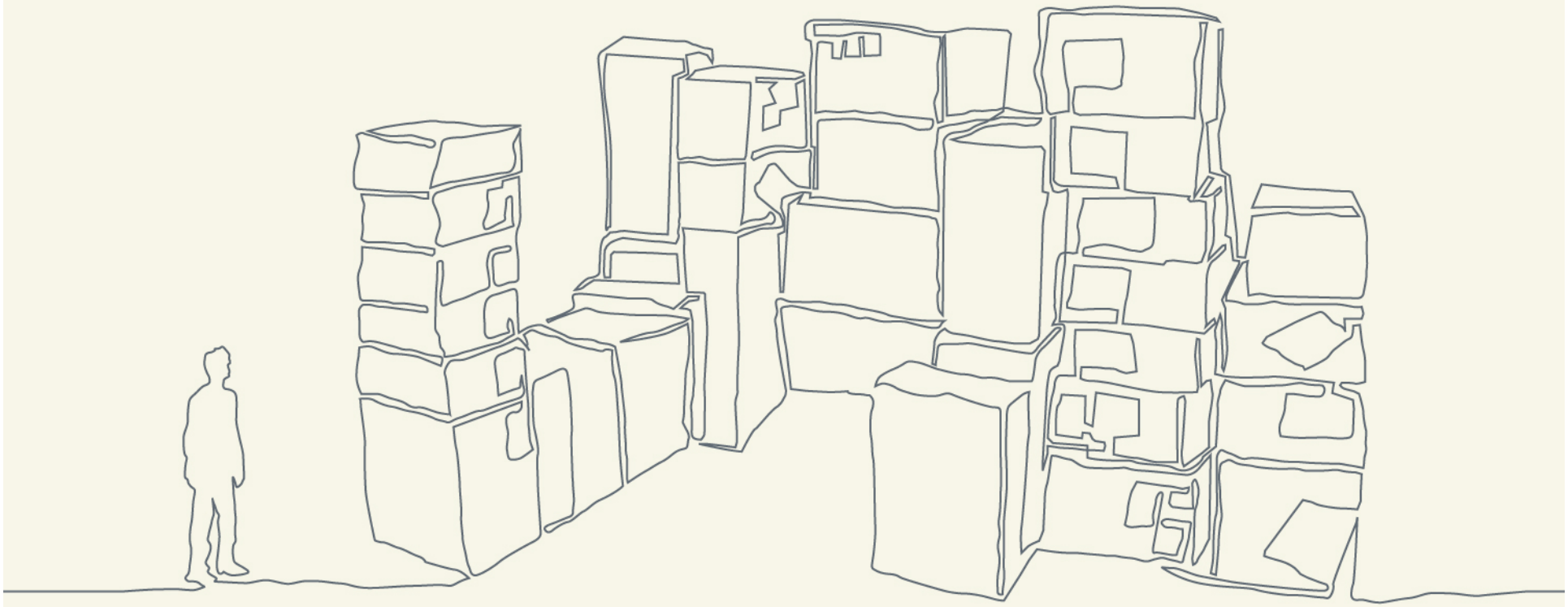
Objectives



Objectives

- Describe in depth the Java 2 Security Architecture
- Show how to use security tools provided by Java Platform
- Show how to use the most common APIs to develop secure applications

Introduction

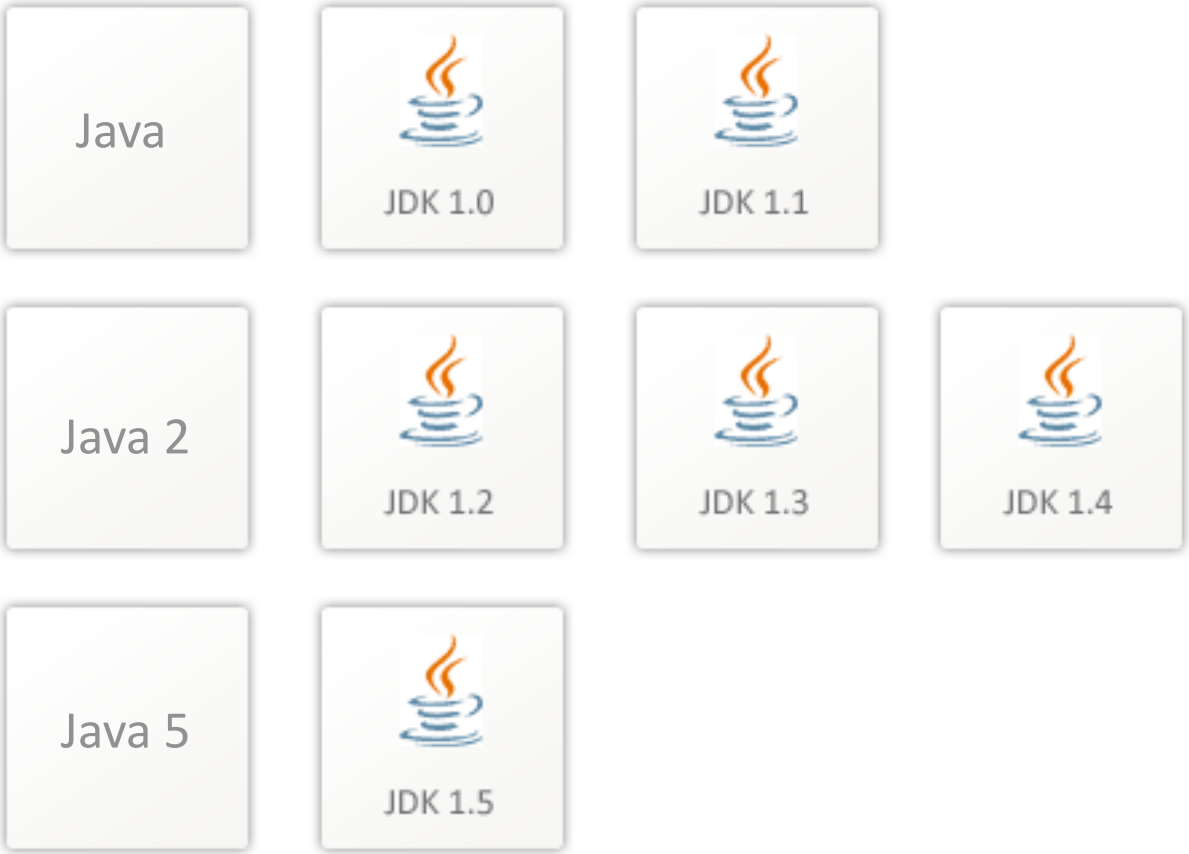


The Java Language

- Developed by Sun Microsystems
- First public release in mid 90s
- Simple, robust, object orientated
- Platform Independent
- Interpreted
- Type-safe
- Garbage Collection

Versions of the Java Language [1 of 2]

Previous versions:



Versions of the Java Language [2 of 2]

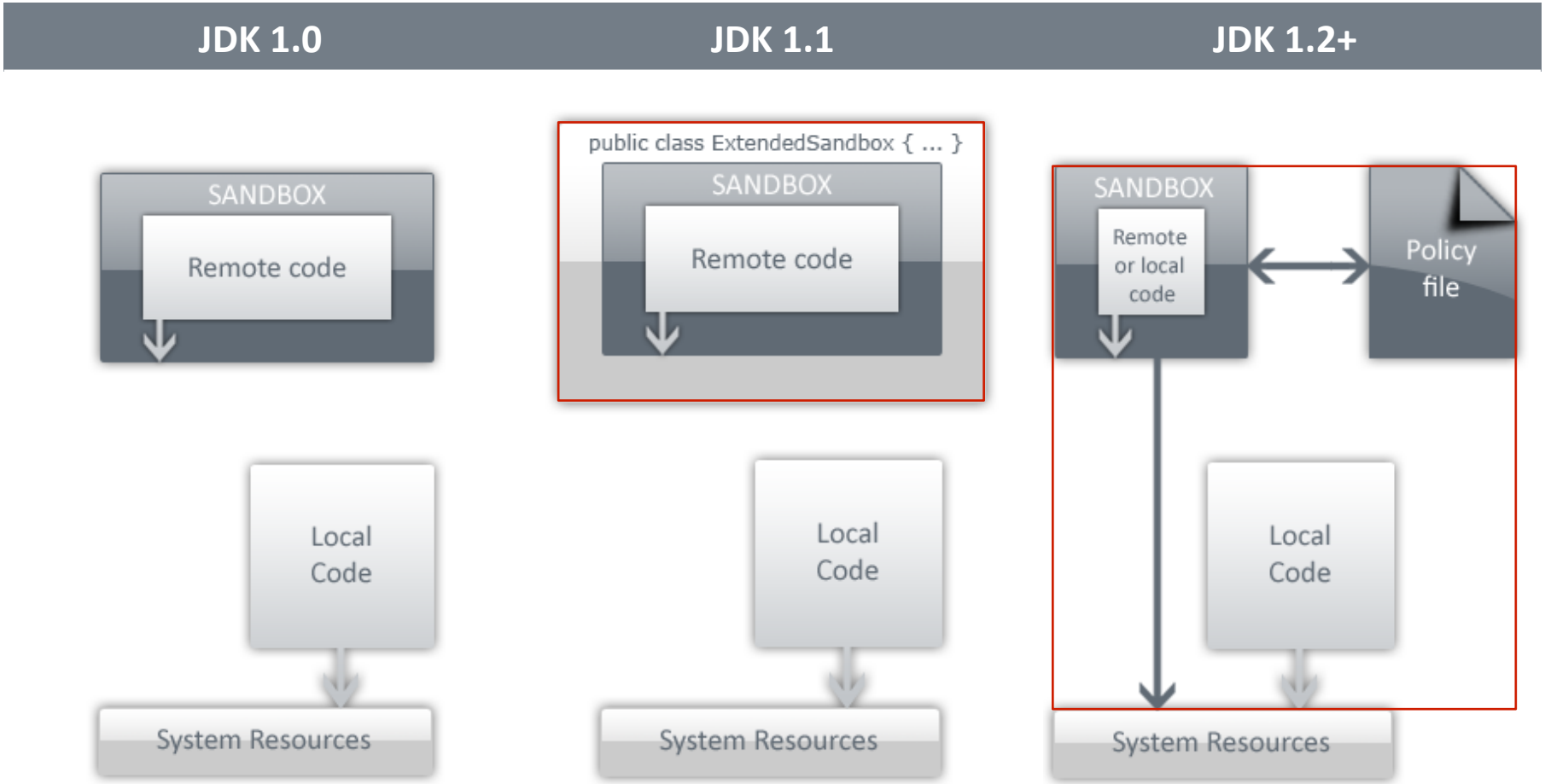
Current version:

Coming soon:

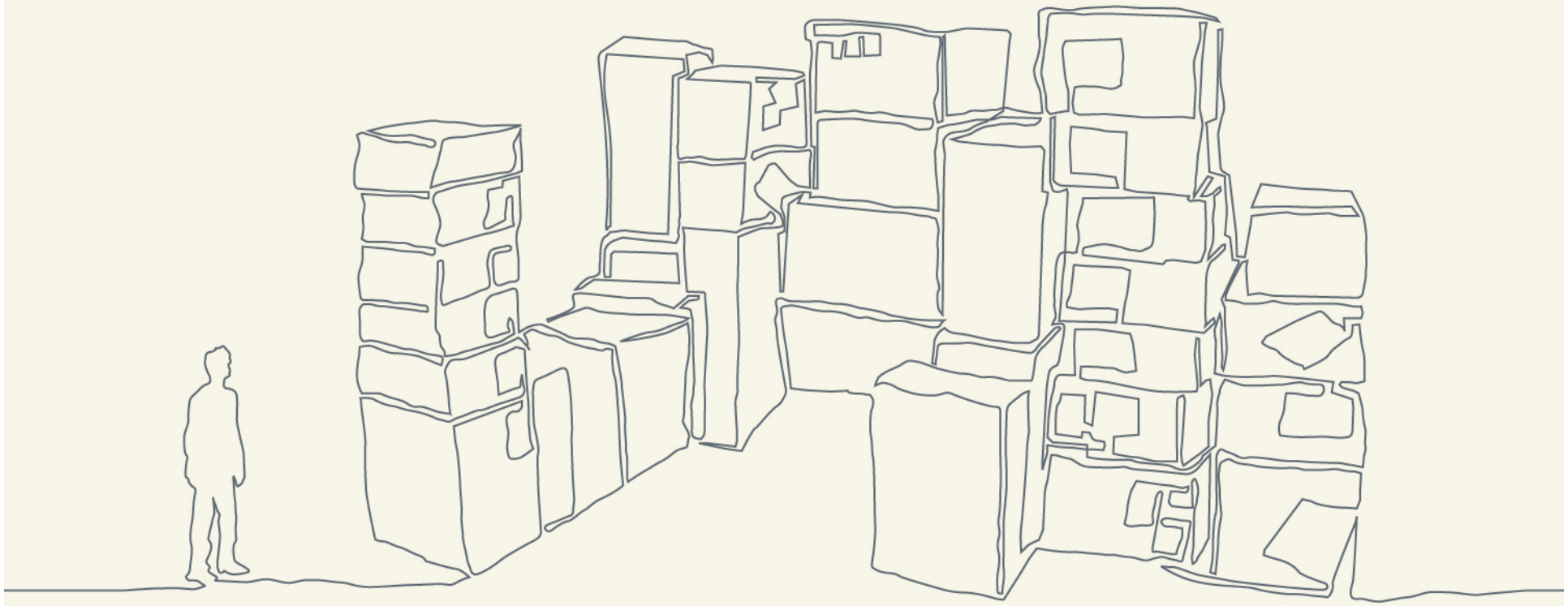
What's the difference? [1 of 2]

JDK 1.0	JDK 1.1	JDK 1.2+
"Black and white" model	"Shade of grey" model	"Fine-grained" model
Trust local code	Trust local code	Trust local code
Don't trust remote code	Don't trust remote code	Don't trust remote code
Remote code in sandbox	Remote code in sandbox	Remote code in sandbox – local code can be too
Sandbox = limited access to system resources	Sandbox can be extended by writing custom code	Sandbox can be extended using a *security policy*
Cannot provide remote code with custom or additional permissions	Applets can be *signed.* Signed code is granted the same permissions as local code	Applets can be signed. Permissions can be fine-tuned by specifying the signers of the code.

What's the difference? [2 of 2]



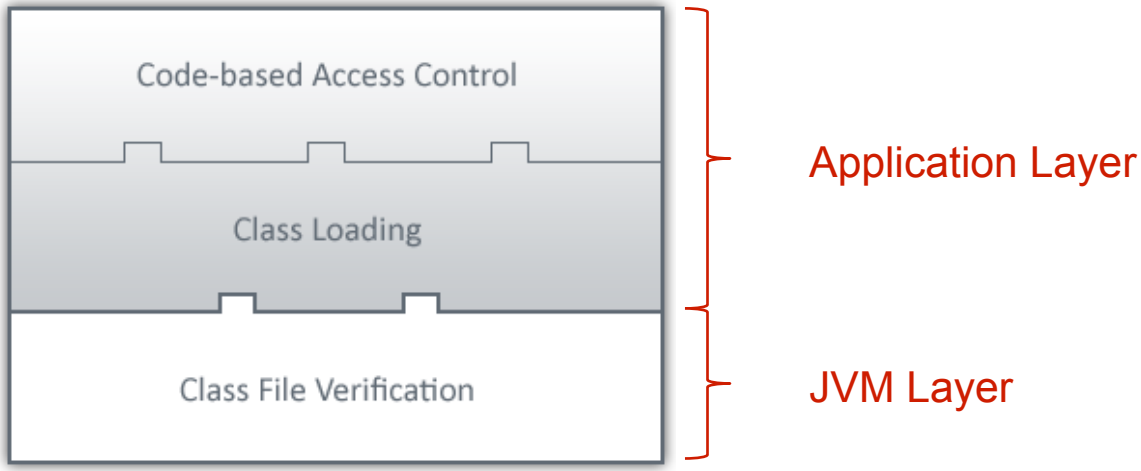
Foundations of Java 2 Security



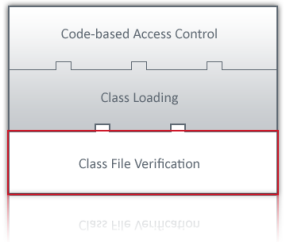
Foundations [1 of 2]

- Security can be split in two layers:
  - Low-level security [JVM layer]: garbage collection, class file verification, ...
  - High-level security [Application layer]: sandbox, security policy, security APIs, ...
- Provides a solid basis for secure applications.
- Secure applications can implement security functions using security APIs [JCE, JSSE, JAAS]

Foundations [2 of 2]

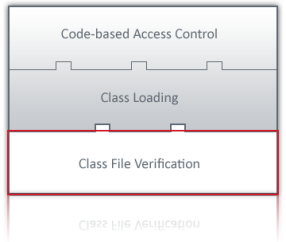


JVM Layer



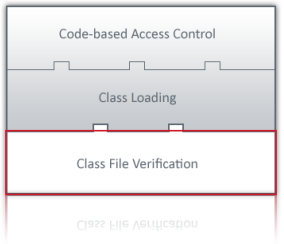
JVM Layer – Java Virtual Machine

- Java Virtual Machine [JVM]:
  - is responsible for platform independence
  - is an abstract computing machine
  - has an instruction set
  - can manipulate memory at runtime
  - does *__not__* interpret Java source code, but
  - Interprets *Java Bytecodes*
  - Java Bytecodes are stored in a *class file*



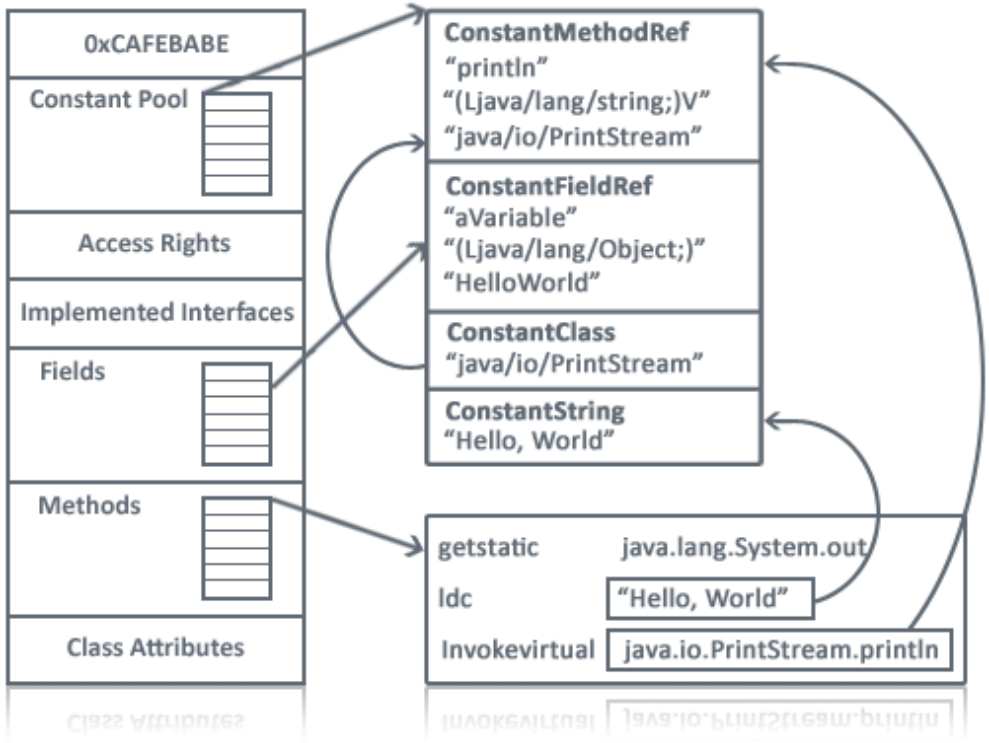
JVM Layer – Class file structure [1 of 2]

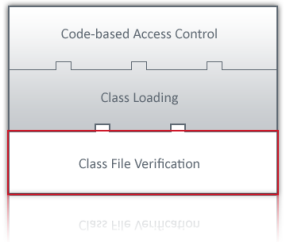
- The class file is made of:
  - A “magic” constant: 0XCAFEBABE
  - Major/minor version information
  - Access flags
  - The “constant pool”
  - Information about the current class (name, superclass, ...)
  - Information about the fields and methods in the class
  - Debugging information



JVM Layer – Class file structure [2 of 2]

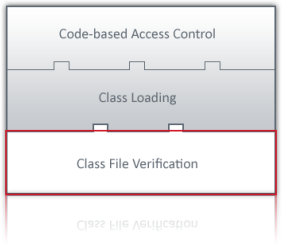
```
public class HelloWorld {
    public static void main(String[] args) {
        String aVariable = "Hello, World";
        System.out.println(aVariable);
    }
}
```





JVM Layer – The Class File Verification [1 of 8]

1. Basic checks on the class file structure
2. Basic checks on the “looks” of:
   - Class references
   - Field references
   - Method references
3. Bytecode verification
4. Actual verification of:
   - Class References
   - Field access/modifications and method calls



JVM Layer – The Class File Verification [2 of 8]

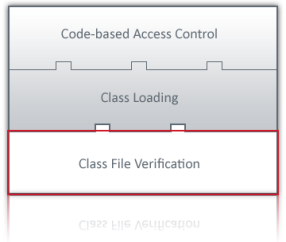
1. Basic checks on the class file structure
   a. First **4 bytes** must equal to 0XCAFEBA
   b. All recognized attributes must have the length
   c. The class file can't be truncated or have at the end
   d. All data in the constant pool must be recognized

Why CAFEBABE?!

1.Need a number to uniquely identify the class file

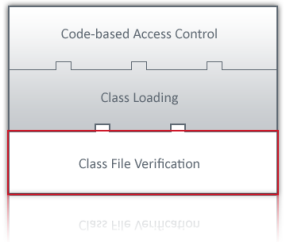
2.Better than:

- **A FAB CAFE**
- **A BAD CAFE**
- **CAFE A FAD**
- **DEAD CAFE**
- **CAFE FACE**
- **...?**



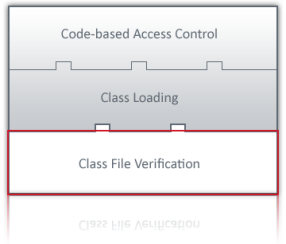
JVM Layer – The Class File Verification [2 of 8]

1. Basic checks on the class file structure
    a. First **4 bytes** must equal to 0XCAFEBABE
    b. All recognized attributes must have the appropriate length
    c. The class file can't be truncated or have extra bytes at the end
    d. All data in the constant pool must be recognized



JVM Layer – The Class File Verification [3 of 8]

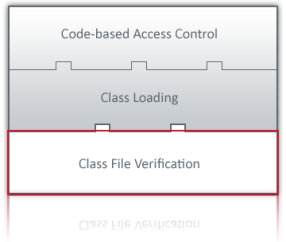
2. Basic checks on the “looks” of:
    a. Class references:
        - `final` classes must not be sub-classed
        - Every class must have a `super` class
    b. Field/method references must have:
        - Legal names/classes
        - Legal type signature
    c. The constant pool satisfies certain constraints



JVM Layer – The Class File Verification [4 of 8]

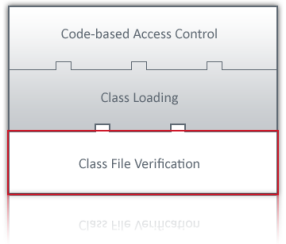
3. Bytecode verification

   - Each instruction is converted into an array
   - The array contains attributes and arguments (if any) for that instruction
   - A flag indicates whether an instruction needs to be verified
   - An algorithm is run on each of the instructions



JVM Layer – The Class File Verification [5 of 8]

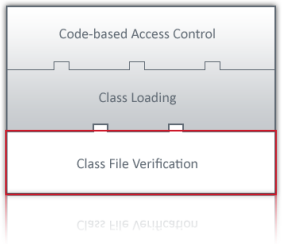
3. Bytecode verification
    1. Find an instruction that needs looking at
    2. Emulate the effect of the current instruction on the stack/registers
        - Check if the elements needed from the registers/stack are of the right type and that there are enough elements on the stack
        - Check if there is enough space for new elements to be placed onto the stack and indicate their type.
        - If registers are modified, indicate those registers contain the new type(s)



JVM Layer – The Class File Verification [6 of 8]

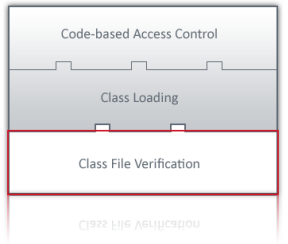
3. Bytecode verification
    3. Determine what instruction will follow this one:
        A. The next instruction [if the current instruction is not a `goto`, `return` or `throw`]
        B. The target of a (un)conditional statement
        C. Exception handler for the current instruction
    4. Merge the stack and registers into each of the successor instructions
    5. Back to step 1

JVM Layer – The Class File Verification [7 of 8]



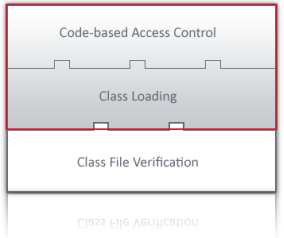
4. Actual verification of:
    1. Class references
        1. Load the class being referred to
           (if not already loaded by previous instructions)
        2. Check if the current class can refer to the referred class
    2. Field access/modifications and method calls
        1. Check if the method or field exists in the given class
        2. If it does, check its type
        3. Check if the current method can access the given field/ method

JVM Layer – The Class File Verification [8 of 8]



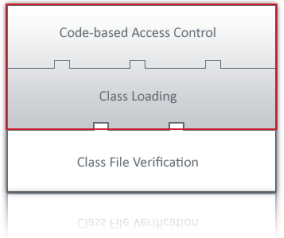
- If successfully verified, the instructions will be flagged
- The JVM will thus not run the verifier on those instructions again

Application Layer



Application Layer – Class loading [1 of 3]

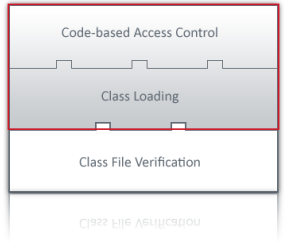
- A **class loader** loads classes on demand
- Referred to as “Lazy Loading” [cf. class file verification – pass 4]
- Reduces memory usage
- Improves system response time
- Enforces **type-safety** alongside the JVM



Application Layer – Class loading [2 of 3]

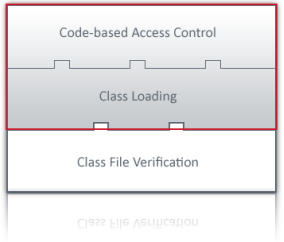
- Instances of class loaders
  - Primordial/bootstrap class loader
    [*loads java base classes*]
  - System class loader
    [*loads classes in the classpath*]
  - Application class loader
    [*defined by the developer*]

Application Layer – Class loading [3 of 3]



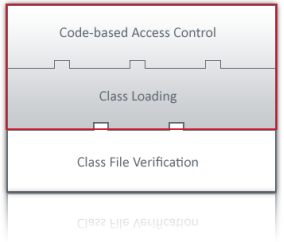
- Class loaders are called by the JVM
- Class loaders are responsible in defining the classes they load and providing a **namespace**
- Class loaders define classes by loading them and associating those to a **protection domain**

Application Layer – Protection Domain [1 of 2]



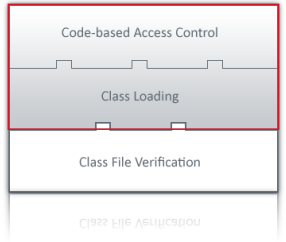
- A **protection domain** associates permissions to class with regards to their location, certificate and *principals*
- A protection domain is **defined by the security policy**
- *Classes that belong to the same protection domain are loaded by the same class loader*
- Similarly, code coming from the same code source belong to the same domain

Application Layer – Protection Domain [2 of 2]



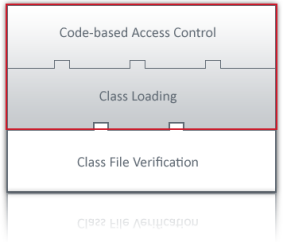
- All Java base classes have a protection domain referred to as the **system domain**
- Any class that does not belong to the system domain belongs to the **application domain**
- The Java Runtime maintains a mapping of classes to their domains
- Similarly, class loaders maintain a *cache* of protection domains for reuse if classes are loaded from a known code source

Application Layer – Code Source



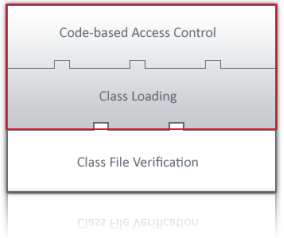
- A code source is simply the URL from where a class is loaded
- A code source may contain a certificate if it has been signed

Application Layer – Security Policy [1 of 2]



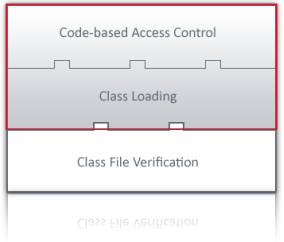
- The **security policy** defines the protection domain
- The *default* security policy is referred to as the **system-wide policy** and implements the default policy for the sandbox
- *Additional* policies are **user-defined**
- You can have *one or more* security policies
- All the references to the security policy (or policies) are specified in the *security property* file located in the installation directory of the JDK (or JRE)

Application Layer – Security Policy [2 of 2]



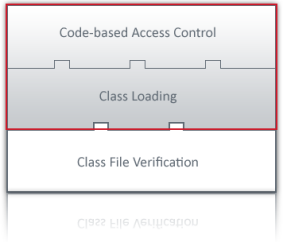

```
/* repository for certificates */
keystore "keystore";

/* policy statements */
/* grant any code signed by phil and running from  */
/* the URL given below read access to C:\demo.txt  */
grant signedBy "Phil", codeBase "http://
www.heathwallace.com/phil" {
    permission java.io.FilePermission "C:/demo.txt",
"read";
 };

/* grant anyone everything (!) */
grant {
    permission java.security.AllPermission;
 };
```

Application Layer – Security Manager

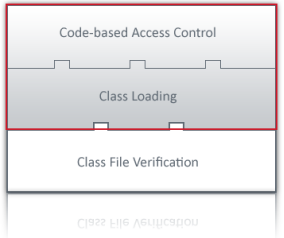


- The **security manager** is the first point of permission checking if system resource access is needed
- Still exists for historical reasons
- All the permission checking are now delegated to the **access controller**

Application Layer – Access Controller [1 of 4]



- The **access controller** *enforces* the security policy by checking whether a piece code has got the required permission to access system resources
- The access controller runs a **stack walking algorithm** on the *caller* stack called the *execution stack*
- The execution stack keeps track of all caller classes

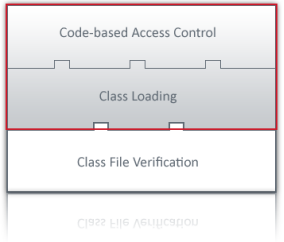


Application Layer – Access Controller [2 of 4]

- The algorithm is as follows:

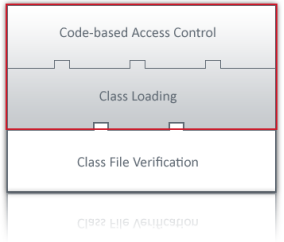
```
i = m;

 while (i > 0) {

      if (caller i's domain does not have the
permission)
                throw AccessControlException

      else if (caller i is marked as privileged) {
                if (a context was specified in the
call to doPrivileged)

context.checkPermission(permission)
                return;
      }
      i = i - 1; /*next method on stack */
 };
```

Application Layer – Access Controller [3 of 4]



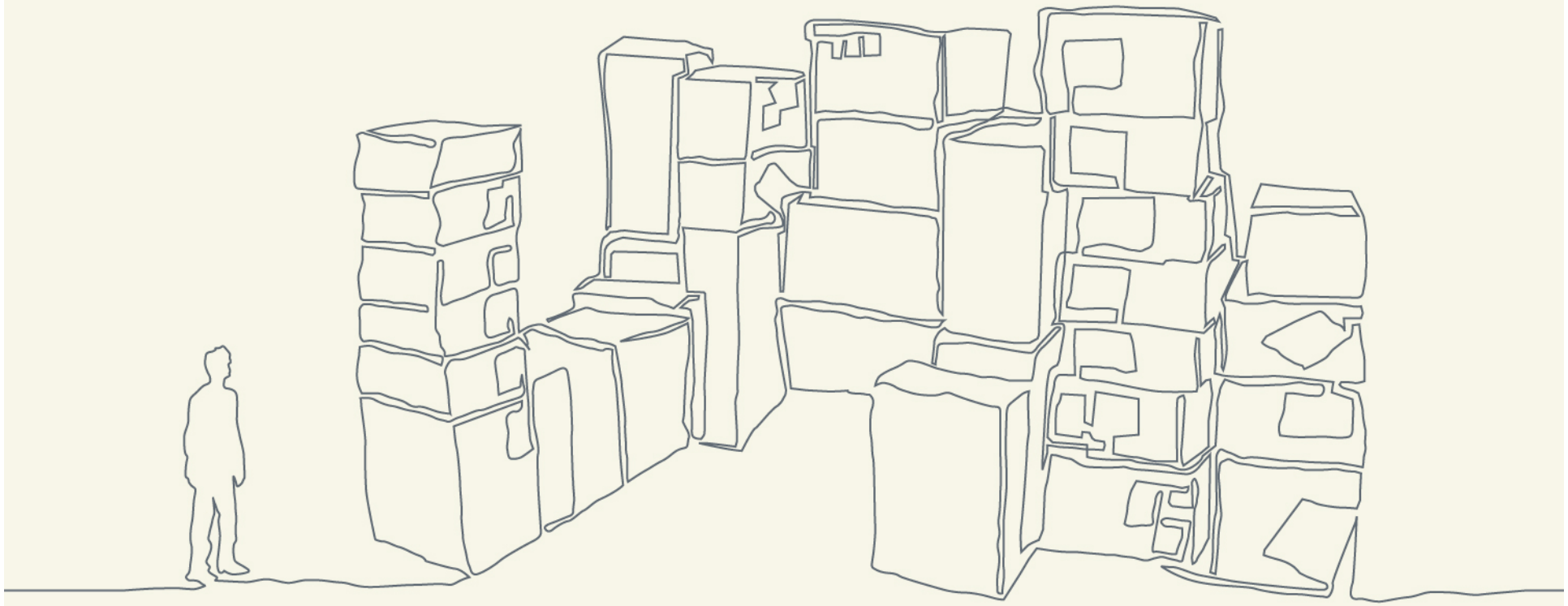
- Start with the most recent class that invoked a *checkPermission()*
- If all the classes have the appropriate permissions with regards to their protection domain, the algorithm returns "silently"
- If a class along the stack is marked as **privileged**, the algorithm returns
- Other classes higher in the stack may or may not have the permissions

Application Layer – Access Controller [4 of 4]



- Classes marked as *privileged* do not gain additional permissions
- The same privileged classes do not “transfer” those privileges to “less powerful” methods/classes

Tools and APIs



Tools

Keytool [1 of 3]

- Command line utility used to
  - Generate private/public key pairs
  - Import/export certificates
- Key pairs and certificates are stored in a physical repository referred to as the “keystore”

Keytool [2 of 3]

- How to
  - Generate a key pair:

```
keytool -genkey –alias <alias> -keystore <keystore> -storepass <password>
```

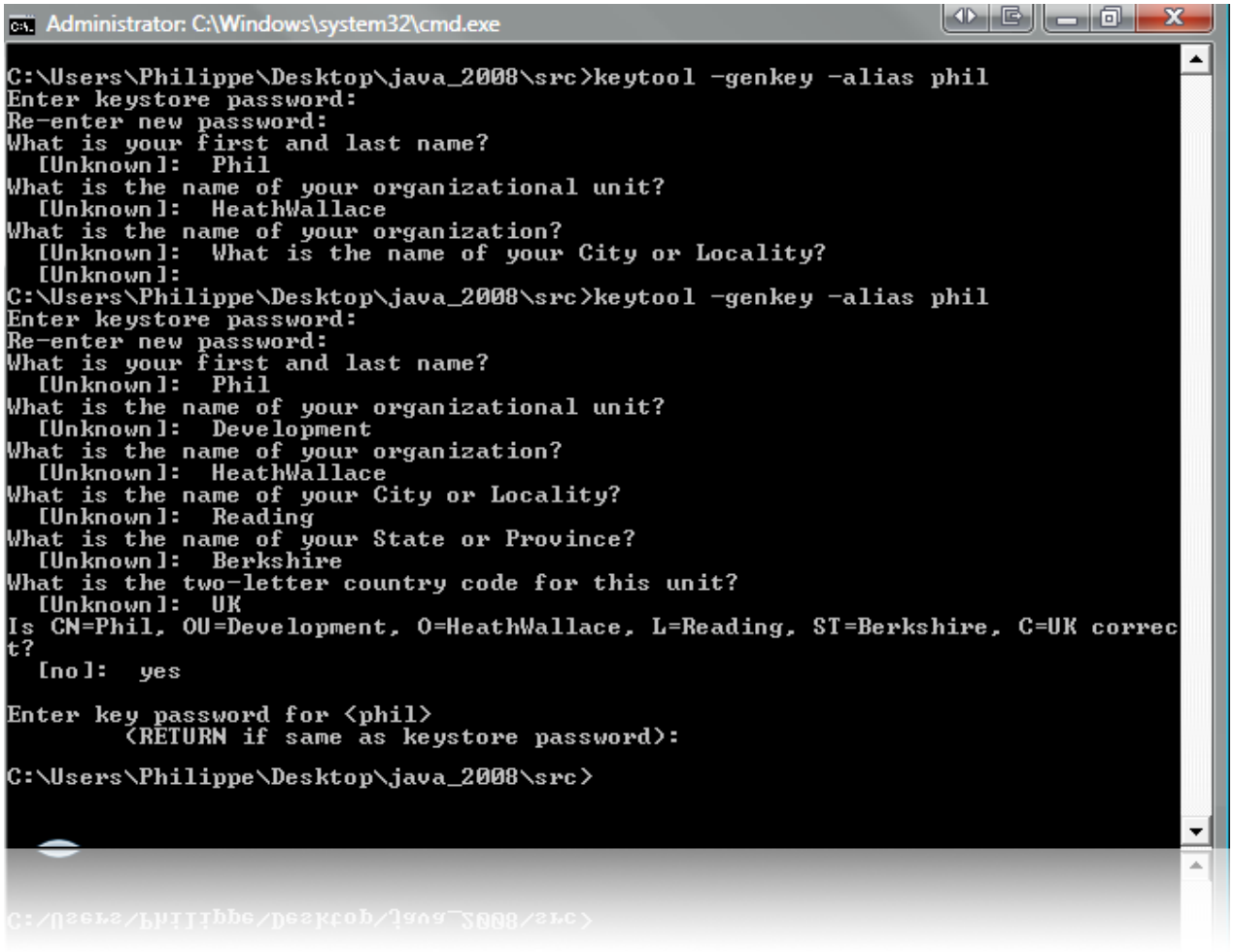
  - Exporting certificate from the keystore:

```
keytool -export –alias <alias> -file <certificate_file>
```

  - Importing certificate to the keystore

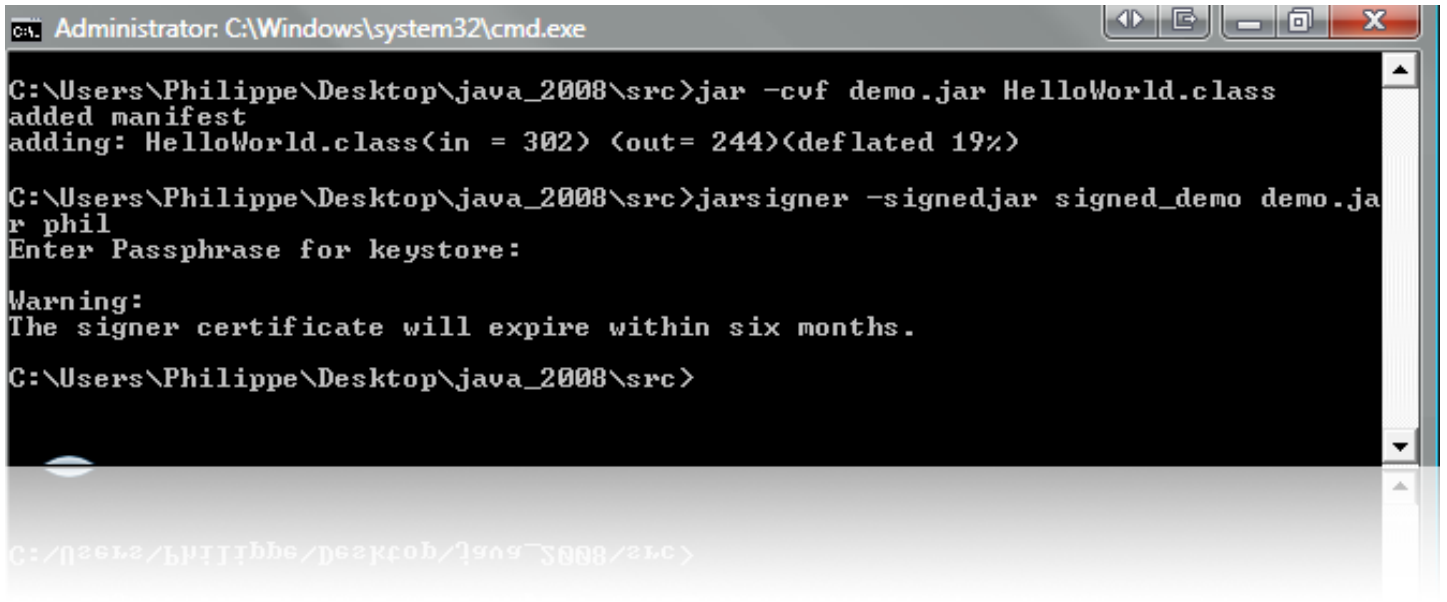
```
keytool -import –file <certificate_file>
```

Keytool [3 of 3]



Jarsigner

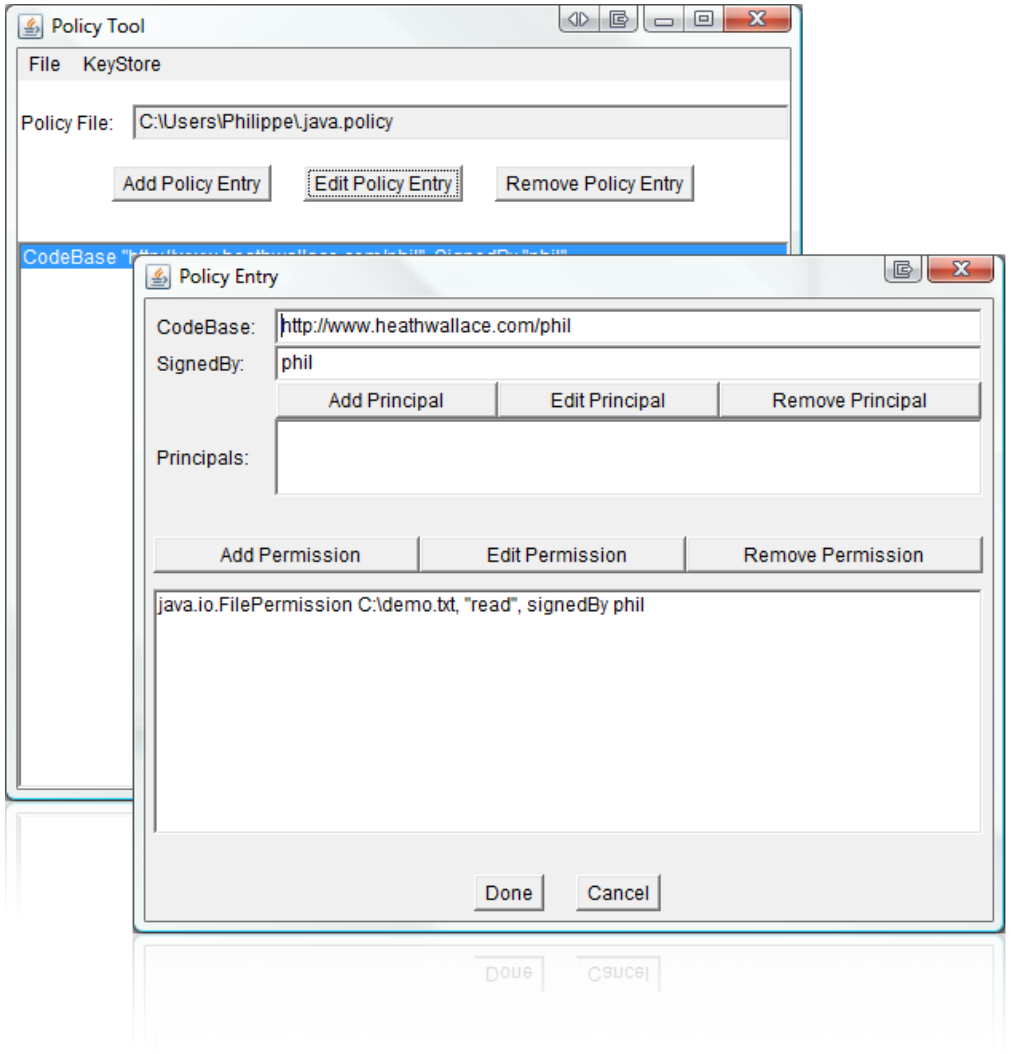
- Command line utility used to sign code
  1. Compress the class files using the jar tool
  2. Sign the java archive (*.jar)



Policytool [1 of 2]

- GUI to editing security policy
  - Generate private/public key pairs
  - Import/export certificates
- Key pairs and certificates are stored in a physical repository referred to as the “keystore”

Policytool [2 of 2]

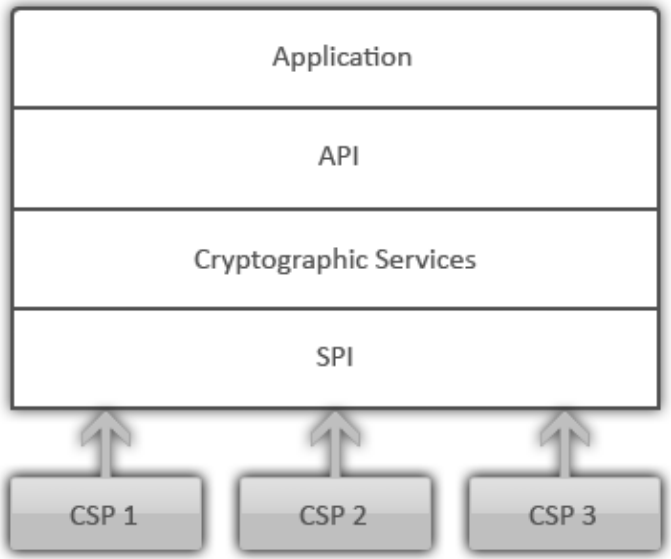


APIs

Java Cryptography Architecture [1 of 5]

- Integral to the platform since JDK 1.4
- Previously referred to as the Java Cryptography Extension (JCE) due to U.S. Regulations on the export of cryptography
- Aims to provide developers a crypto API to use without being concerned with the implementation of algorithms

Java Cryptography Architecture [2 of 5]



Source: Inside Java 2 Platform Security

Java Cryptography Architecture [3 of 5]

- **Engine class**: defines a *crypto service* in an abstract manner without providing a concrete implementation
- **Service Provider Interface (SPI)** provides the *crypto interface* to the application via the engine
- Each engine class has a corresponding SPI which defines exactly what crypto method a **Cryptographic Service Provider (CSP)** must *implement*

Java Cryptography Architecture [4 of 5]

- Symmetric Encryption Algorithms
  - DES - default keylength of 56 bits
  - AES
  - RC2, RC4 and RC5
  - IDEA
  - Triple DES – default keylength 112 bits
  - Blowfish – default keylength 56 bits
  - PBEWithMD5AndDES
  - PBEWithHmacSHA1AndDESede
  - DES ede

Modes of encryption:

- ECB
- CBC
- CFB
- OFB
- PCBC

Source: OWASP - http://www.owasp.org/index.php/Using_the_Java_Cryptographic_Extensions

Java Cryptography Architecture [5 of 5]

- Asymmetric Encryption Algorithms
  - RSA
  - Diffie-Hellman – default keylength 1024 bits
- Hashing / Message Digest Algorithms
  - MD5 – default size 64 bytes
  - SHA1 – default size 64 bytes

Source: OWASP - http://www.owasp.org/index.php/Using_the_Java_Cryptographic_Extensions

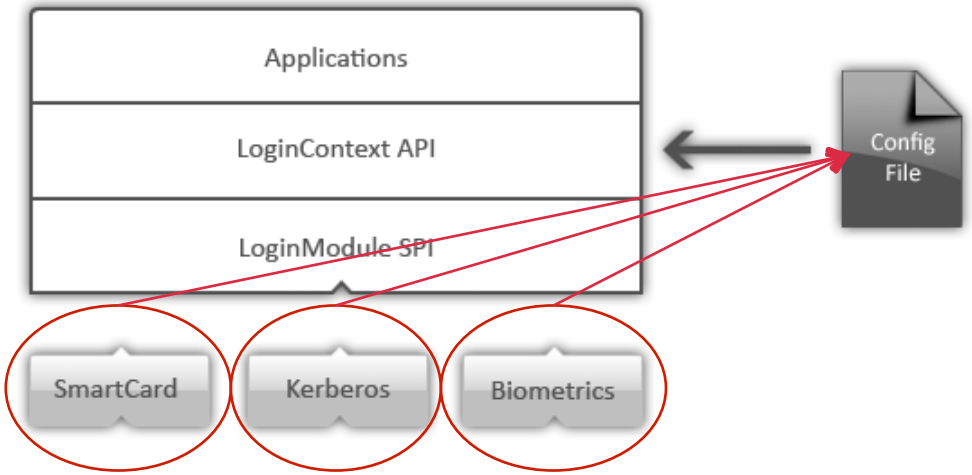
Java Authorization and Authentication Services [1 of 7]

- Java Authorization and Authentication Services (JAAS) provides user-based access control
- Thus extends the current code-based policy model
- It is possible to combine both mechanisms in the security policy

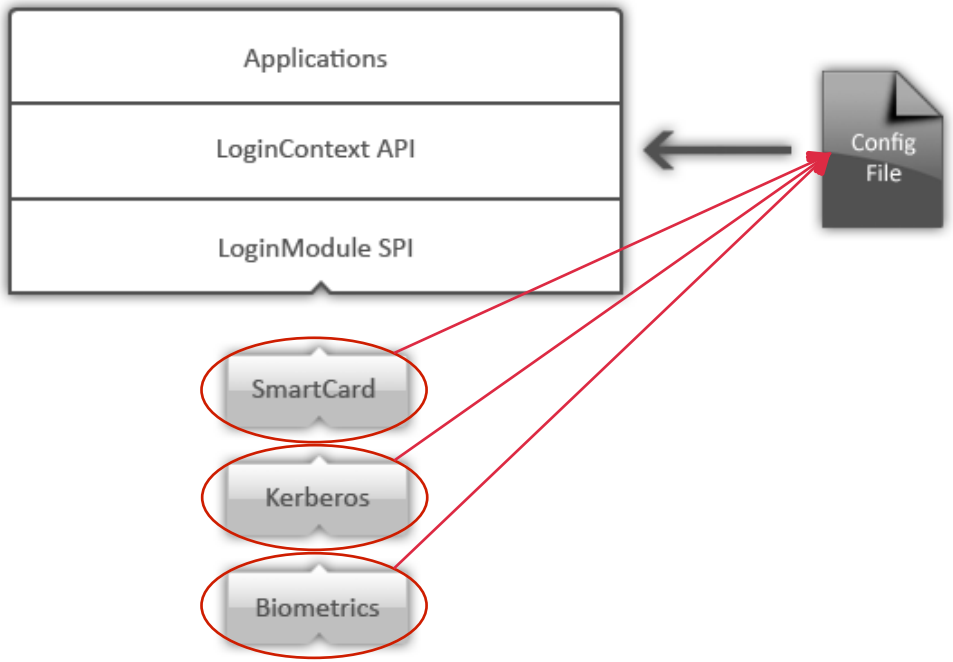
Java Authorization and Authentication Services [2 of 3]

- A **subject** is an entity that wishes to authenticate to a service
- A **principal** is the association between a *name* and a *subject*
- A subject can have *multiple names* for different services
- A subject thus has a set of *multiple principals*

Java Authorization and Authentication Services [3 of 7]



Java Authorization and Authentication Services [3 of]



Java Authorization and Authentication Services [4 of 7]

- A **login context** provides the basic methods to authenticate a subject
- The login context uses the **configuration file** to determine which **login module** to use
- The configuration file can contain multiple login modules

Java Authorization and Authentication Services [5 of 7]

- The login context performs authentication in *two steps*
  1. Login context invokes a login module to verify a subject's identity
  2. Login context invokes a login module to commit to the authentication process
- The login context the subject to its relevant principal

Java Authorization and Authentication Services [6 of 7]

- The login context performs authentication in *two steps*
  1. Login context invokes a login module to verify a subject's identity
  2. Login context invokes a login module to commit to the authentication process
- The login context the subject to its relevant principal

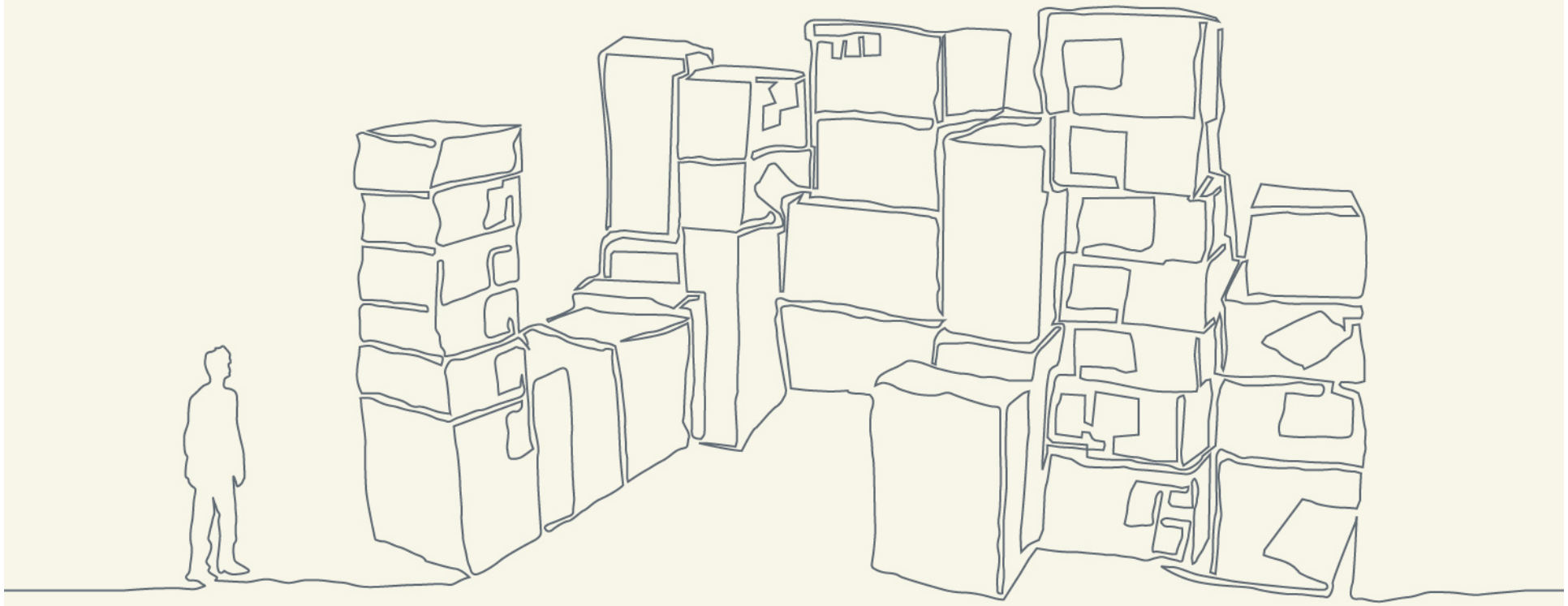
Java Authorization and Authentication Services [7 of 7]

- Authorization is *principal-based* **not** subject-based
- Thus, permissions are granted to a subject *based on the authenticated principal it contains*

Java Secure Socket Extension

- Provides a set of APIs and implementations for SSL version 3 and TLS version 1
- Trust establishment through certificates in keystores
- Keystore on server side is referred to as “keystore”
- Keystore on client side is referred to as “truststore”

Conclusion



Conclusion [1 of 5]

1. Java is rich-featured platform
2. Provides secure foundation to building more secure software
    1. Class file verifier
    2. Protection domain
    3. Code-based access control
    4. Privileged code execution
    5. Security policy

Conclusion [2 of 5]

3. JVM Layer – Class file verification ensures:
    1. There are no stack overflow or underflow
    2. All register accesses and stores are valid
    3. All bytecode instruction parameters are valid
    4. There is no illegal data conversion

Conclusion [3 of 5]

4. Application layer
    1. Class loaders define classes and associate them with a protection domain
    2. The protection domain is comprised of:
        - A code source (URL, certificate)
        - A set of permissions
        - A set of principals

Conclusion [4 of 5]

3. Protection domain is defined by the security policy
4. The security policy is enforce by the access controller
5. The access controller runs a stack walking algorithm to determine if classes have the appropriate permissions

Conclusion [5 of 5]

5. Provides security packages/libraries for developers to build security-related applications
    1. JCA to provide crypto functionality
    4. JAAS to provide support for authentication/ authorization
    5. JSSE to provide implementations of SSL/TLS

Resources

1. Class File Structure
    - http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html
2. Bytecode verification
    - http://gallium.inria.fr/~xleroy/publi/bytecode-verification-JAR.pdf

Resources

3. Java 2 Security

   - Inside Java 2 Platform Security, Second Edition – **ISBN**: **0201787911** or **978-0201787917**
   - http://java.sun.com/javase/technologies/security/index.jsp
   - Hacking Exposed - J2EE & Java – ISBN: **0072225653** or **978-0072225655**

5. Practical examples

   - http://java.sun.com/docs/books/tutorial/security/index.html

Thank you!