Security notes.

General.

- nothing is 100% secure
- only as strong as the weakest link (e2e security requires many layers)
- manageable (a complex system will only serve to confuse admins/users)
- security must be included as part of the design not retro-fitted

Concepts.

<u>Threats.</u>

- secrecy (access to confidential information)
- integrity (information is altered in transit)
- availability (denial of service)

<u>Protections.</u>

- perimeter defences / shields (e.g. firewall)
- VPN

<u>Cryptography / crypto-analysis.</u>

Algorithms which facilitate provision of :

- confidentiality (protect information from unauthorized use / third parties)
- integrity (protect information from alteration by third parties)
- authentication (verify the *Principal* is who they claim to be via their *Credentials*)
- non-repudiation (sender cannot deny they sent information)

<u>Policy / access control.</u>

Restrict access by use of a policy – policy is separate from the mechanism (so as to allow easy configuration of the policy for different environments/situations etc.)
Access may restrict code execution (e.g. file writes not allowed) or enforce access based on *who* is executing the code.

N.B. JDK1.2 policy is totally flexibly – the policy file can be set up so that an applet from a remote, unsigned JAR will execute; a standalone app can be restricted so that local file read/write is disallowed.

Principals are mapped to *Roles*, *Permissions* are grouped and associates with Roles.

Cryptography.

One-way hash (a.k.a. Message Digest)

- like a fancy checksum
- fixed length output
- quick to compute
- hard to determine original from hash
- hash output is a function of the message input – i.e. different for each message
- **NOT** based on keys
- Examples : MD4, MD5, SHA
- one-way hash + key == Message Authentication Code (MAC), e.g. HMAC

Cipher.

Apply cipher to plain text to get cipher (encrypted) text.
Two types :

- symmetric cipher (one key for encryption/decryption) – e.g. DES
- asymmetric cipher (one key for encryption/one for decryption and vice versa – i.e. public/private keys)

Digital Signatures.

Provides :

- proof* of authenticity of the sender
- integrity of the message
- non-repudiation
- confidentiality (optional, sender encrypts using recipients public key; recipient is the only one that can decrypt the message using their private key)
- Examples : DSA, RSA

For performance reasons, the signature is usually a signed hash – i.e. generate a hash of the message and sign it using the senders private key (a.k.a. the *digital fingerprint*).

The signature is generated using the senders private key so provides proof* of authenticity.
The signature is a signed hash of the message so guarantees the integrity of the message – if the compute hash doesn't match either the signature is wrong or the hash is wrong.
As the sender is the only one that knows the private key, they cannot deny that they signed the message.

* For digital signatures to work, its necessary to verify that the senders public key is authentic. Also, even if the sender is authentic, how does the recipient know they can be trusted ?
Certificates and *Certificate Authorities* fulfil this requirement.

Certificates.

A certificate authority (*CA* or *issuer*) issues certificates to certificate *subjects* (a.k.a. *owners*) after a stringent verification process.

The certificate contains the subject's public key, the issuer's digital fingerprint (a.k.a. *thumbprint*), a validity period and *distinguished name* (*Common Name, Operational Unit, Organisation, Location, State, Country*) fields for the issuer and subject.
If the subject and issuer are the same, the certificate is *self-signed*.

Consequently, if a user trusts the CA they can verify certificates issued by the CA by verifying the digital signature. If the signature verifies and the certificate hasn't expired, the certificate is valid.
In some cases a *certificate chain* is used – certificate was issued by CA2 whose certificate was issued by CA1 whose certificate was issued by CA.

Public key certificates are stored in the standards based X.509 format (X.503v3 is the latest standard).

Java Security.

Language support.

Java is type-safe and catches common security traps (e.g. buffer overflow) at compile time.

JVM.

The runtime environment support security :

- handles unchecked exceptions (e.g. null pointer, out of memory)
- garbage collection (ensures memory integrity)
- byte code verification for user class files
- the ClassLoader screens user class files at an early stage
- a SecurityManager can apply the security policy

JDK1.0 – the "sandbox"

Applets are untrusted and are confined to the *sandbox* – i.e. no access to local file system, only open network connections on the server the applet came from, cannot read certain system properties (user/java dirs, username), cannot start a new process, cannot load libraries or define native methods, can't exit JVM, etc.
Applications – trusted by default, allowed to do anything

JDK1.1 – all or nothing

Applets can be signed and packaged into JAR files. Providing the public key of the applet signer is trusted, applets are trusted and can do anything an application can do.
Applications – trusted by default, allowed to do anything

Jars can be signed using the `jarsigner` tool.

JDK1.2 – fine grained security

Applets are not trusted by default, applications are trusted by default.
However, both can be subjected to a security policy.

The policy is organised by *Protection Domains* – contains a set of *Permissions* for a *CodeSource* (URL of *CodeBase* + the associated public keys). Consequently, each application / applet runs in the protection domain associated with the location of the code being executed.
N.B. JDK1.2 policy is totally flexibly – the CodeBase and SignedBy can vary independently such that code can come from anywhere, not be signed, trust code any code signed by X, etc.
E.g. set up so that an applet from a remote, unsigned JAR will execute; a standalone app can be restricted so that local file read/write is disallowed.

The security *Policy* is the set of Protection Domains specified in a policy file (a custom file or `JAVA_ROOT/jre/lib/security/java.policy`)
The policy can be updated using `policytool` (provided with the JRE)

The default *SecurityManager* (abstract before JDK1.2) applies the policy. The default implementation delegates to the *AccessController*

The *Keystore* is a database of certificates and private/public keys referenced by an *Alias*.
The Keystore can be updated using `keytool` (provided with the JRE)
To update the keystore, you require the keystore password. Private keys may also be protected by passwords.
The default keystore is stored in `JAVA_ROOT/jre/lib/security/cacerts`

JAAS (Java Authentication and Authorisation Service).

Adds permission based on who is executing the code.

JCE (Java Cryptography Extension).

Provides support for encrypting / decrypting information – DES, Blowfish, etc.