Introduction to Java Cryptography

Java Support for Cryptography

- Java Cryptography Architecture (JCA)
  - Separate concepts, algorithms, and implementations
  - *Concepts*: e.g. ciphers, signatures, message digests, …
  - *Algorithms*: e.g. RSA, DSA, MD5, …
  - *Implementations*: supplied by different cryptographic providers, e.g. Sun
  - **`java.security.*`**
- Java Cryptography Extension (JCE)
  - Supports encryption/decryption
  - **`javax.crypto.*`**
  - Separate from JCA for historical reasons

Algorithms and Implementation Providers

- Different algorithms for same cryptographic concept
  - E.g. for “cipher” there are DES, AES, RSA, …
- Different implementations for the same algorithm
  - Provided by different providers
  - Default provider: SUN / SunJCE
- General idea of Java cryptography: *encapsulation*
  - Algorithm/implementation details hidden from programmers
  - Only need to know the concepts

Cryptographic Concept Classes

- Java provides many cryptographic objects
  - Example: a **`Cipher`** object
    - This is not an algorithm, nor the ciphertext, but rather an object that captures the generic cipher operations
- These objects are created using *factory methods*
  - **`getInstance()`**
  - A static method (i.e. class method), does not need an instance to run. Invoked directly from a class (similar to **`Math.sqrt()`**, **`Integer.valueOf()`**, …)
  - Returns an instance (i.e. object) of a class
  - E.g. **`MessageDigest.getInstance("MD5")`** is invoked on the **`MeesageDigest`** class directly, to create an object of class **`MessageDigest`**

Keys in Java

- Recall: what is a key?
  - A parameter to a cipher (e.g. the value *n* in rot-*n*)
  - A numeric value, or equivalently, a bit string
  - Different values of key gives different encryption results
- Key size:
  - Usually long (e.g. 512 bits) for security (prevent brute-force attack)
  - Too long will make program slow
- In Java, a key is represented by an *interface*
  - Hides implementation detail

The `java.security.Key` interface

- All keys have
  - An algorithm: the algorithm for which the key works
  - An encoded form: a representation as a byte array
  - A format: the format of representation, e.g. X.509
- Three methods in the `Key` interface:
  - `String getAlgorithm()`
    - Returns the name of algorithm for which this key is used
  - `byte[] getEncoded()`
    - Returns the encoded value of the key
  - `String getFormat()`
    - Returns the name of the encoding format

Extensions of the `Key` Interface

- Specific interfaces defined for secret keys or public/private keys
  - For clarity and type safety only; no new methods
  - `javax.crypto.SecretKey`
  - `java.security.PublicKey`
  - `java.security.PrivateKey`
- Key pairs: represented by `java.security.KeyPair`
  - `KeyPair(PublicKey publicKey, PrivateKey privateKey)    // constructor`
  - `PublicKey getPublic()`
  - `PrivateKey getPrivate()`
    - Returns public and private keys, respectively, from a key pair

Key Generators

- How to create a key?
- Step 1: Obtain a key generator object for the specific algorithm
  - `java.security.KeyPairGenerator` (for public/private key pairs)
  - `javax.crypto.KeyGenerator` (for secret keys)
- Step 2: Initialize the key generator
  - E.g. `abstract void initialize(int strength, SecureRandom sr)`
- Step 3: Ask the key generator to generate a key (or key pair)
  - E.g. `abstract KeyPair genKeyPair()`
  - E.g. `abstract SecretKey generateKey()`

More on Key Generators

- Examples:
  - A 1024-bit key pair for RSA

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance(“RSA”);
kpg.initialize(1024);
KeyPair kp = kpg.genKeyPair();
```

  - A DES key

```
KeyGenerator kg = KeyGenerator.getInstance(“DES”);
kg.init(new SecureRandom());
SecretKey key = kg.generateKey();
```

  - Note: public (**`KeyPairGenerator`**) and private (**`KeyGenerator`**) key generators have different method names, e.g. **`initialize`**() vs. **`init`**()

Algorithm Initialization

- *Overloaded* initialization methods for **`KeyGenerator`**:
  - **`static final void init(SecureRandom rand)`**
  - **`static final void init(int strength)`**
  - **`static final void init(int strength, SecureRandom rand)`**
  - Strength: usually is length of key
  - Similarly for **`KeyPairGenerator`** (see API for details)

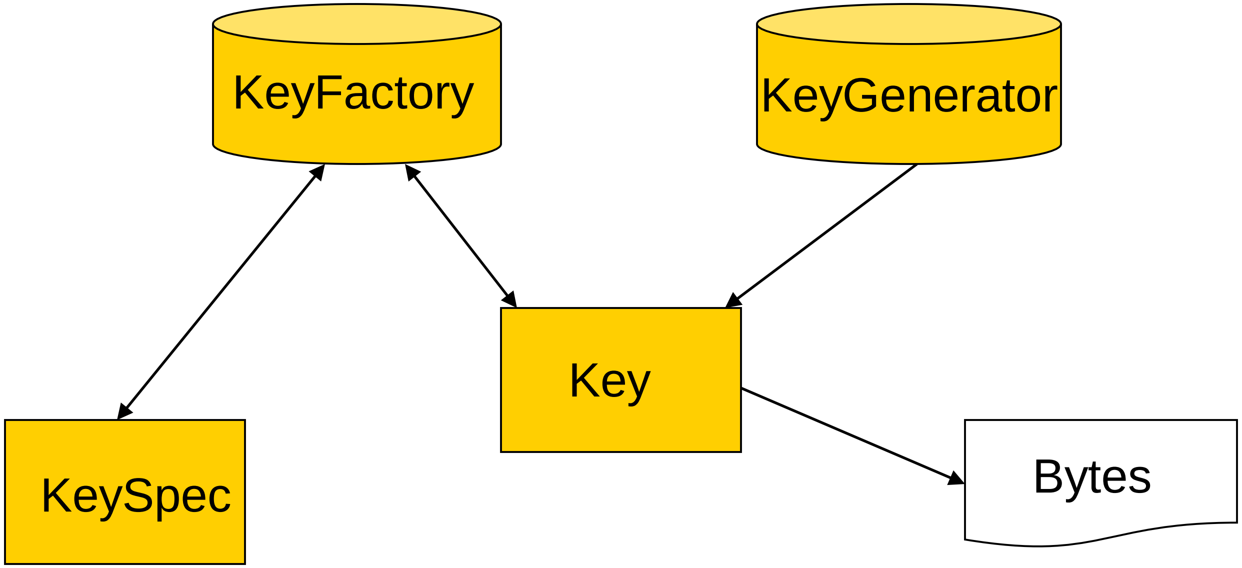
- Secure random number generation
  - Computers are bad at generating truly random numbers
  - **`java.util.Random`**: not secure enough
  - **`java.security.SecureRandom`**: a more secure PRNG (PseudoRandom Number Generator)

Key Specifications

- A Java Key object can only give you the encoded value (bytes) of the key; but often a key is constructed out of some *key materials*
  - E.g. for RSA, public key = (e, n), private key = (d, n)
- A **`KeySpec`** is an interface for the specification of such data
  - There are subclasses for individual algorithms
- A **`KeyFactory`** convert between Key and KeySpec
- Also allows the key materials to be extracted / stored (e.g. in a file)
  - Another method: Serialization (object streams)

Key Factories

- A **`KeyFactory`** converts a **`Key`** object to/from its specification (a **`KeySpec`**)
  - (No, this is not a factory for generating keys.)
- Relationships:



`(Secret-)KeyFactory` Classes

- **`KeyFactory`** is for public/private keypairs, **`SecretKeyFactory`** is for secret keys
- General ways of using them:
  - Obtain the object:
    - Use the **`getInstance(String algorithm)`** methods
  - Create an appropriate **`KeySpec`**
  - From things to keys:
    - Then call **`generateSecret()`** or similar methods
  - From keys to things:
    - Use **`getKey()`** or similar methods
- Use cipher-specific subclasses of these classes/interfaces

Examples

- Can also use cipher-specific methods, e.g. RSA:

```
RSAPublicKey r = (RSAPublicKey)pubkey;
RSAPrivateKey s = (RSAPrivateKey)privkey;
BigInteger n = r.getModulus();
BigInteger e = r.getPublicExponent(); // (e,n) of pub key
BigInteger d = s.getPrivateExponent(); // (d,n) of priv key
System.out.println(“n=” + n);
System.out.println(“e=” + e);
System.out.println(“d=” + d);
```