Java Security

Prof. Dr. Marc Rennhard, Dr. Stephan Neuhaus
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema | neut @zhaw.ch

This topic is partly based on the security chapter of this textbook: *Cay S. Horstmann and Gary Cornell, Core Java Volume 2. Advanced Features, Prentice Hall International.* The book chapter is also available online at *http://www.informit.com/articles/article.aspx?p=1187967.*

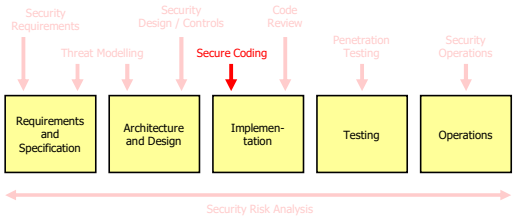
For general information about Java security, refer to the official Java security developer's guide: *https://docs.oracle.com/en/java/javase/11/security/index.html*.

Content

- Java Cryptography Architecture (JCA)
  - Provides the functionality to carry out basically any cryptographic operation in Java
- Java Secure Sockets Extension (JSSE)
  - Provides the functionality to implement TLS protected communication channels in Java
- Optional appendix with further examples

Goals

- You understand the basic principle how cryptographic algorithms are provided by the Java Cryptography Architecture (JCA)
- You can use these algorithms to implement cryptographic operations in your programs in an efficient and secure way
- You can use the functionality offered by the Java Secure Sockets Extension (JSSE) to implement TLS protected communication in Java
- You can use the *keytool* to create key pairs and certificates
- Security activity covered in this chapter:



Java Cryptography Architecture (JCA)

Java Cryptography Architecture (1)

- JCA is a component of Java SE and provides functionality to perform a variety of cryptographic operations, including:
  - Hash functions (MD5, SHA-1/2/3), message authentication codes (HMAC)
  - Secret and public key cryptography (AES, 3DES, RC4, CHACHA, RSA, ECC)
  - Diffie-Hellman key exchange, pseudo random number generation,...
- JCA uses a provider-based architecture
  - This means that the actual implementations of the cryptographic algorithms are provided by different Cryptographic Service Providers (CSP) in a plug-in manner
  - Java SE includes several CSPs per default, which means most cryptographic algorithms are available in Java SE «out of the box»
  - In addition, there exist some CSPs that are provided by 3rd parties
    - Usually, CSPs from 3rd parties are only used if Java SE does not support a specific cryptographic algorithm you want use (e.g., SEED block cipher is currently not supported in Java SE)
    - One of the most prominent 3rd party provider of CSPs is the Legion of the Bouncy Castle (*www.bouncycastle.org*)

Cryptographic Service Provider (CSP)

Basically, a CSP is a software component (i.e., a library) that can be plugged into the JCA and that contains classes that provide the functionality of one or more cryptographic algorithms.

The main advantage of this plugin-based architecture is that it can be extended with cryptographic algorithms that are not part of standard Java SE, and these algorithms can then be used in the same way as the included algorithms.

Java Cryptography Architecture (2)

- To use a CSP from a 3rd party, it must be added to the configuration file *$JAVA_HOME/conf/security/java.security*:

```
# List of providers and their preference orders (see above):
security.provider.1=SUN
security.provider.2=SunRsaSign
...
security.provider.14=org.bouncycastle.jce.provider.BouncyCastleProvider
```

  - The file also includes the CSPs that are part of Java SE
- In addition, the CSP jar file from the 3rd party provider must be included in the classpath when running the program, e.g.:
  - *java -cp bcprov-jdk15on-168.jar:. ...*
- When creating a crypto object, e.g., one of type *MessageDigest*, one can optionally specify the provider

```
md = MessageDigest.getInstance("SHA-256");
md = MessageDigest.getInstance("SHA-256", "BC");
```

  - If no provider is specified, the highest-priority provider that provides the algorithm is used (which is usually used in practice)

Bouncy Castle

To make use of Bouncy Castle in your application, you need to:

- Download the BouncyCastleProvider jar file
- Add the following line to *$JAVA_HOME/conf/java.security*:
  security.provider.<Priority>=org.bouncycastle.jce.provider.BouncyCastleProvider
- Make sure the jar file is included in the classpath when running the program

Java Cryptography Architecture (3)

- The API provided by JCA is extensive and contains many classes and methods
- Here, we look at the following:
  - Computing a hash with various algorithms
  - Generating random numbers, secret keys and public key pairs
  - Secret key encryption
  - Public key encryption and signatures
  - This should give you a good idea how to use the JCA
    - Additional information can be found in the API documentation of the JCA classes and in the official Java security developer's guide
- Remember Kingdom 3: Security Features?
  - Even when using the predefined security functions of a programming language, it's still easy to make mistakes, e.g., using short key lengths, re-using initialization vectors, using wrong cipher modes, using insecure random number generators
  - Therefore, it's important you understand how to use a crypto library correctly and the JCA serves as an example to do this

Computing a Hash (1)

Computing a hash (message digest) in Java requires the following steps:

- Create the desired *MessageDigest* object

```
MessageDigest hashFunction = MessageDigest.getInstance("SHA3-256");
```

  - *MessageDigest* is a factory class to create an object that can compute a specific message digest
  - That object is itself a subclass of *MessageDigest*

- To perform hash computations, the *MessageDigest* class provides the methods *update* and *digest*

- *update* feeds individual bytes or byte arrays into the *MessageDigest* object (but does not yet compute any output)

```
InputStream in = ...;
int bt;
while ((bt = in.read()) != -1)
  hashFunction.update((byte) bt);
```

```
byte[] input = ...;
hashFunction.update(input);
```


Conversion of int to byte

in.read() returns the next byte from the input stream as data type int. The value is between 0 and 255. The *update* method requires a byte type and if the int value is directly passed to it, there would be a compilation error. By explicitly converting it to byte, this compilation error is prevented. Note that there is no loss of precision, as the int value – in this case – is always between 0 and 255 and can therefore safely be converted to byte.

Computing a Hash (2)

- *digest* is used to compute and return the hash

```
// Computes the hash over the data that was fed into hashFunction
byte[] hash = hashFunction.digest();
```

  - In addition, *digest* can also feed additional bytes into the *MessageDigest* object before the hash is computed

```
// First feeds additional data (input) into hashFunction and then
// computes the hash over all data that was fed into hashFunction
byte[] hash = hashFunction.digest(input);
```

  - *digest(input)* can also be used without any previous call to *update* to simply compute the hash over input

- Question: What's the difference between the two computed hashes?

```
hashFunction.update((byte) 'H');
hashFunction.update("ello".getBytes());
byte[] hash = hashFunction.digest();
```

```
byte[] hash = hashFunction.digest
              ("Hello".getBytes());
```

No difference, they are equal

Computing a Hash – Example (1)

hash gets the byte array to hash and a hash algorithm to use and as parameters and returns the hash value as a hex string

main receives the string to hash as a command line parameter and computes hashes using different hash algorithms

```
public class Hash {
  public static String hash(byte[] input, String algorithm) {
    try {
      MessageDigest hashFunction = MessageDigest.getInstance(algorithm);
      byte[] hash = hashFunction.digest(input);
      return Util.toHexString(hash);
    } catch (Exception e) {
      return ("" + e);
    }
  }

  public static void main(String[] args) {
    List<String> algorithms = new ArrayList<>(
      Arrays.asList("MD5", "SHA1", "SHA-256", "SHA-512", "SHA3-256",
                    "SHA3-512", "SHA4-512"));
      for (String algorithm : algorithms) {
        System.out.println(algorithm + ": " + hash(args[0].getBytes(),
                           algorithm) + "\n");
      }
    }
  }
}
```


Util.toHexString

This is a helper method to convert a byte array to a hex string:

```
package util;
public class Util {
  public static String toHexString(byte bytes[]) {
  StringBuilder sb = new StringBuilder();
  for (byte b : bytes) {
      sb.append(String.format("%02X", b));
  }
  return sb.toString();
  }
}
```

Computing a Hash – Example (2)

- Running the example generates the following:

```
clt-mob-t-6208:classes marc$ java hash/Hash Test
MD5: 0CBC6611F5540BD0809A388DC95A615B

SHA1: 640AB2BAE07BEDC4C163F679A746F7AB7FB5D1FA

SHA-256: 532EAABD9574880DBF76B9B8CC00832C20A6EC113D682299550D7A6E0F345E25

SHA-512: C6EE9E33CF5C6715A1D148FD73F7318884B41ADCB916021E2BC0E800A5C5DD97F5142178F6AE88C8
FDD98E1AFB0CE4C8D2C54B5F37B30B7DA1997BB33B0B8A31

SHA3-256: C0A5CCA43B8AA79EB50E3464BC839DD6FD414FAE0DDF928CA23DCEBF8A8B8DD0

SHA3-512: 301BB421C971FBB7ED01DCC3A9976CE53DF034022BA982B97D0F27D48C4F03883AABF7C6BC778AA
7C383062F6823045A6D41B8A720AFBB8A9607690F89FBE1A7

SHA4-512: java.security.NoSuchAlgorithmException: SHA4-512 MessageDigest not available
```

MD5 and SHA1 are NOT secure!

- This shows that Java (just like any other crypto library) includes several algorithms for backward compatibility that should not be used anymore!

If an algorithm is used that is not available, a *NoSuchAlgorithmException* is thrown (other crypto classes have the same behavior)

Random Number Generation (1)

- To generate cryptographically secure random numbers (which can be used for keys), Java provides the class *SecureRandom*
  - Attention: There's also a class *Random* (in *java.util*), but *Random* is not intended for cryptographic applications
- Using *SecureRandom* works as follows:

Create a *SecureRandom* object and call its *nextBytes* methods to get the desired number of random bytes

```
public class PRNG {
  public static void main(String[] args) {
    byte[] randomBytes = new byte[32];
    SecureRandom random = new SecureRandom();
    random.nextBytes(randomBytes);
    System.out.println(Util.toHexString(randomBytes));
  }
}
```

- Program output:

```
clt-mob-t-6208:classes marc$ java prng/PRNG
9AB4667FCB442B3170D3332FC08AC569B9833F38841CA13686A83BD5DC8C7802
```

Random Number Generation (2)

- There are two kinds of random number generators: *true* random number generators (TRNG) and *pseudo* random number generators (PRNG)
  - TRNGs are hardware RNGs, which are based on random physical processes
  - PRNGs are deterministic algorithms that create the random numbers based on a seed (must be random and non-predictable) they get as a parameter
- What algorithm does *SecureRandom* use? Answer: several / it depends
  - Per default, it uses the random sources provided by the underlying OS, e.g., */dev/random* or */dev/urandom* on Linux/Unix/macOS-like systems
    - They either use a hardware RNG if available or a PRNG that is seeded with random material collected by the OS (mouse, network, keyboard,...)
  - In addition, general PRNGs such as *DRGB* and *SHA1PRNG* are supported
    - Which are seeded with the random sources provided by the OS
- It's therefore best to create *SecureRandom* as in the previous example, as this uses the random sources of the OS, which are usually good ones
  - But if you want to specify the algorithm, it can be done as follows:

```
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
```


Additional Seeding of *SecureRandom*

It's also possible to feed additional seeding material into a *SecureRandom* object using the method *public void setSeed(byte[] seed)*.

Note that this seed only supplements the already used seeding material and does not replace it. Therefore, repeated calls to this method can never reduce randomness.

Usually, it can be expected the the seeding source (or RNG source) used by *SecureRandom* is «good enough» and therefore, this method is usually not used in practice unless you explicitly want to add your own seeding material for whatever reason.

Secret Key Generation (1)

- Secret keys are required by secret key ciphers and when computing message authentication codes (MAC)
- Creating a new random secret key works as follows (AES, 128 bits):

```
// Create an instance of class KeyGenerator that can create
// specific secret keys (here: AES)
KeyGenerator keyGen = KeyGenerator.getInstance("AES");

// Initialize the generator with the key size (here: 128 bits)
keyGen.init(128);

// Use the generator to create a random secret key
SecretKey key = keyGen.generateKey();
```

The resulting *SecretKey* object can be used to initialize a crypto object such as *Cipher* or *Mac*

  - To create the key, *KeyGenerator* uses *SecureRandom* in its default configuration (i.e., *new SecureRandom()*)
- Alternatively, one can specify the *SecureRandom* configuration to use:

```
SecureRandom random = SecureRandom.getInstance("SHA1PNRG");
keyGen.init(128, random);
```

  - But this is usually not used (unless you explicitly want to do this)

Key and *SecretKey*

Key is an interface and *SecretKey* is a subinterface of *Key*. There are also other subinterfaces for other types of keys, e.g., *PublicKey* and *PrivateKey*. A *Cipher* or *Mac* object requires an object of type *Key* when initializing it, so any of the subinterfaces of *Key* (more specific *Key*s) can be used for initialization.

Specifying the algorithm

Maybe you are wondering why "AES" is needed in *getInstance*, because a secret key is more or less always the same, independent of the actual secret key algorithm that is used. Basically, that's correct, however, by specifying the algorithm, it can be prevented that incorrect key lengths can be generated, which makes using the *KeyGenerator* class and the keys it generates «type safer». For instance, with AES, the *init* method will accept only 128, 192 and 256 as key lengths, as other lengths are not supported by AES. And with other algorithms, there will be other accepted key lengths.

Secret Key Generation (2)

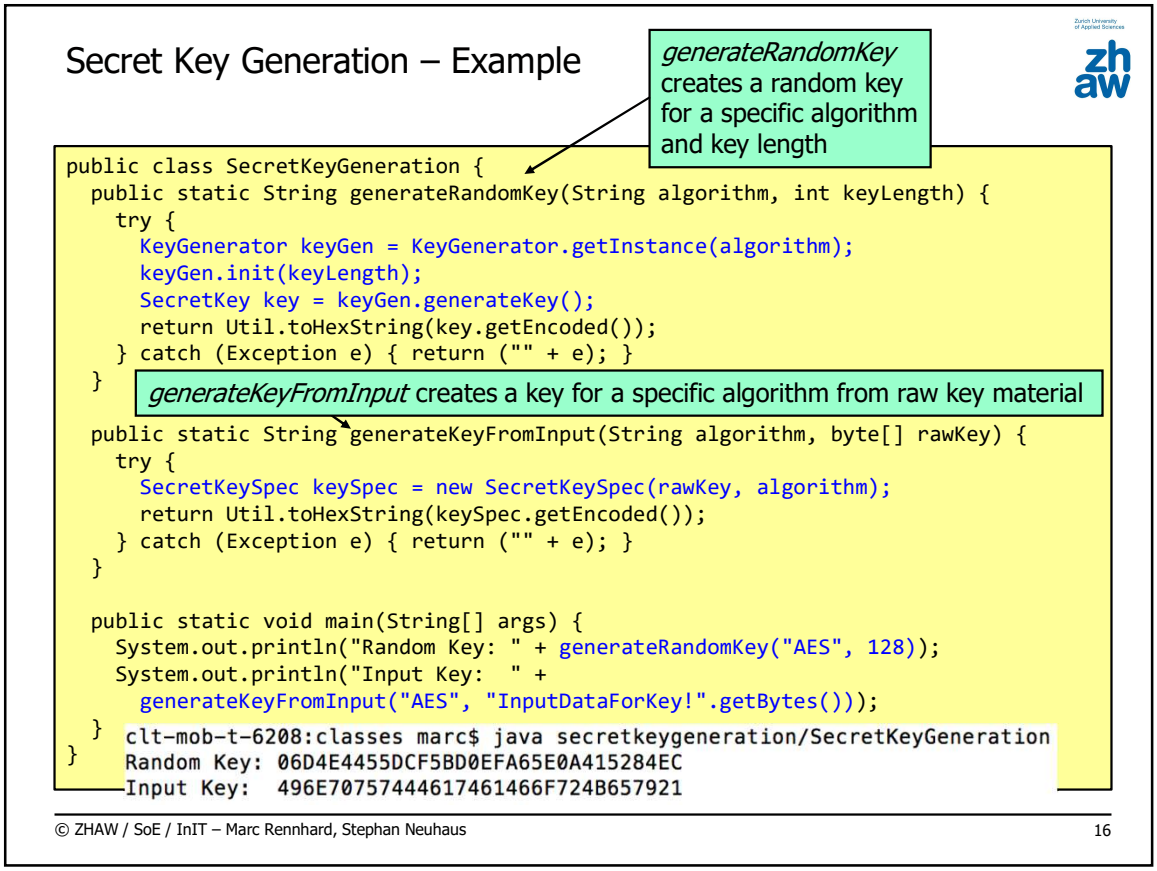
- One can also produce keys from existing «raw key material»
  - E.g., if someone else has given you the key and you are using it to encrypt data for her
- Creating a secret key in this case works as follows:

```
// The raw key material (e.g., 128 bits), available as byte array
byte[] rawKey = ...;

// Generate a SecretKeySpec object from the raw material by
// specifying the type of key (here: AES)
SecretKeySpec keySpec = new SecretKeySpec(rawKey, "AES");
```

The resulting *SecretKeySpec* object can also be used to initialize a crypto object such as *Cipher* or *Mac*

SecretKeySpec

SecretKeySpecKey implements the *SecretKey* interface (which itself extends the *Key* interface), which means it can also be passed to a *Cipher* or *Mac* object to initialize it.



Strength of the Keys

Of course, the key which is based on the string *InputDataForKey!* (to be precise: it is based on the ASCII codes of the characters in the string) is much weaker than the one that is randomly generated and it could easily be broken with a dictionary-based attack in practice. So if you use keys that are based on passwords or other user-selected data, make sure that this data is long and random enough so the resulting key can be considered a strong key.

Secret Key Cryptography (1)

- For secret key encryption / decryption, the class *Cipher* is provided
- Creating a *Cipher* object for a specific algorithm works similar as with *MessageDigest*: by using *getInstance()*

```
Cipher cipher = Cipher.getInstance(algorithm);
```

- The parameter *algorithm* specifies the cipher algorithm, cipher mode and padding scheme, for example:
  - *AES/CBC/PKCS5Padding*: AES in CBC mode with PKCS5Padding
  - *DESede/CTR/NoPadding*: 3DES in CTR mode (CTR doesn't use padding)
  - *AES/GCM/NoPadding*: AES in GCM mode (GCM doesn't use padding)
  - *AES*, *Blowfish*, *DES*: When using a block cipher without mode / padding, it uses per default ECB mode and PKCS5Padding (=*AES/ECB/PKCS5Padding*)
  - *RC4*, *CHACHA20*: Stream ciphers (which have neither mode nor padding)

PKCS5Padding

A commonly used padding scheme is the one described in the Public Key Cryptography Standard (PKCS) #5 by RSA Security Inc. (ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf). In this scheme, the last block is not padded with a pad value of zero, but with a pad value that equals the number of pad bytes. In other words, if L is the last (incomplete) block, then it is padded as follows (we assume a block length of 8 bytes here):

L 01	if length(L) = 7
L 02 02	if length(L) = 6
L 03 03 03	if length(L) = 5
. . .	
L 07 07 07 07 07 07 07	if length(L) = 1

Finally, if the length of the input is actually divisible by 8, then one block

08 08 08 08 08 08 08 08

is appended to the input and encrypted. For decryption, the very last byte of the plaintext is a count of the padding characters to discard.

Source: Cay S. Horstmann and Gary Cornell, Core Java Volume 2. Advanced Feature

Algorithms, Modes and Padding Schemes

For more details about supported cipher, modes and padding schemes, refer to *http://download.oracle.com/javase/1.5.0/docs/guide/security/jce/JCERefGuide.html*

Secret Key Cryptography (2)

- Once the *Cipher* object has been created, it must be initialized by setting the key and the encryption / decryption mode:

```
int mode = ...;
Key key = ...;
cipher.init(mode, key);
```

- There are 4 modes:
  - `Cipher.ENCRYPT_MODE`: for encryption
  - `Cipher.DECRYPT_MODE`: for decryption
  - `Cipher.WRAP_MODE`: to encrypt a key with another key
  - `Cipher.UNWRAP_MODE`: to decrypt a key with another key
  - (the latter two modes are just convenience modes that can be helpful in special cases, e.g., when using hybrid encryption (see appendix))

Secret Key Cryptography (3)

- To encrypt or decrypt data, the methods *update* and *doFinal* are used
  - The usage of these methods depends on whether the input data is processed in a single step or in multiple steps
- When processing all data in a single step, *doFinal* is used and the data to process is passed as an argument

```
byte[] completeInput = ...;
byte[] output = cipher.doFinal(completeInput);
```

- When processing the data in multiple steps, *update* is used as many times as needed and *doFinal* once at the end (with or without data)

```
byte[] input, output;
while (...) { // Loop until all input data is processed
  input = ...; // E.g., read data from an input stream into input
  output = cipher.update(input);
  // Do something with output, e.g., write it to an output stream
}
output = cipher.doFinal();
// Do something with output, e.g., write it to an output stream
```


update

In general, the *update* method can process an arbitrary number of bytes. However, with a block cipher, the output is always a number of blocks, i.e., a multiple of the block length. Any remaining data simply «stays» in the *Cipher* object and is processed together with additional data during a subsequent call of *update* (see explanation later in this chapter). Also, there are different variants of the *update* method, check out the Java API Specifications for details. For instance, there's *update(byte[] input, int inputOffset, int inputLen)* to specify which part of the input data array is actually «inserted» into the cipher object.

***doFinal* with and without Parameters**

As you can see, *doFinal* can contain input data or not as a parameter. If input data is used, then this data is first inserted into the cipher object (i.e., treated as the final chunk of plaintext or ciphertext data) before the encryption or decryption is finally completed.

Just like with update, there are further versions of *doFinal*. E.g., *doFinal(byte[] input, int inputOffset, int inputLen)* allows to specify which part of the input data array is inserted into the cipher object.

Combining *update* and *doFinal*

update and *doFinal* can be combined in many ways. So it's also possible to repeatedly use *update* and then *doFinal* with the last chunk of input data.

Secret Key Cryptography (4)

- In either case, the final step always must be a call of *doFinal* to finish the encryption / decryption operation, for various reasons:
  - When encrypting with a block cipher, it guarantees that the padding is done correctly before encrypting the final block
  - When decrypting with a block cipher, it guarantees that padding is correctly removed after the final block has been decrypted
  - When using modes GCM or CCM that include integrity protection, it guarantees that the authentication tag is computed and appended
- Question: What's the difference between the two computed outputs?

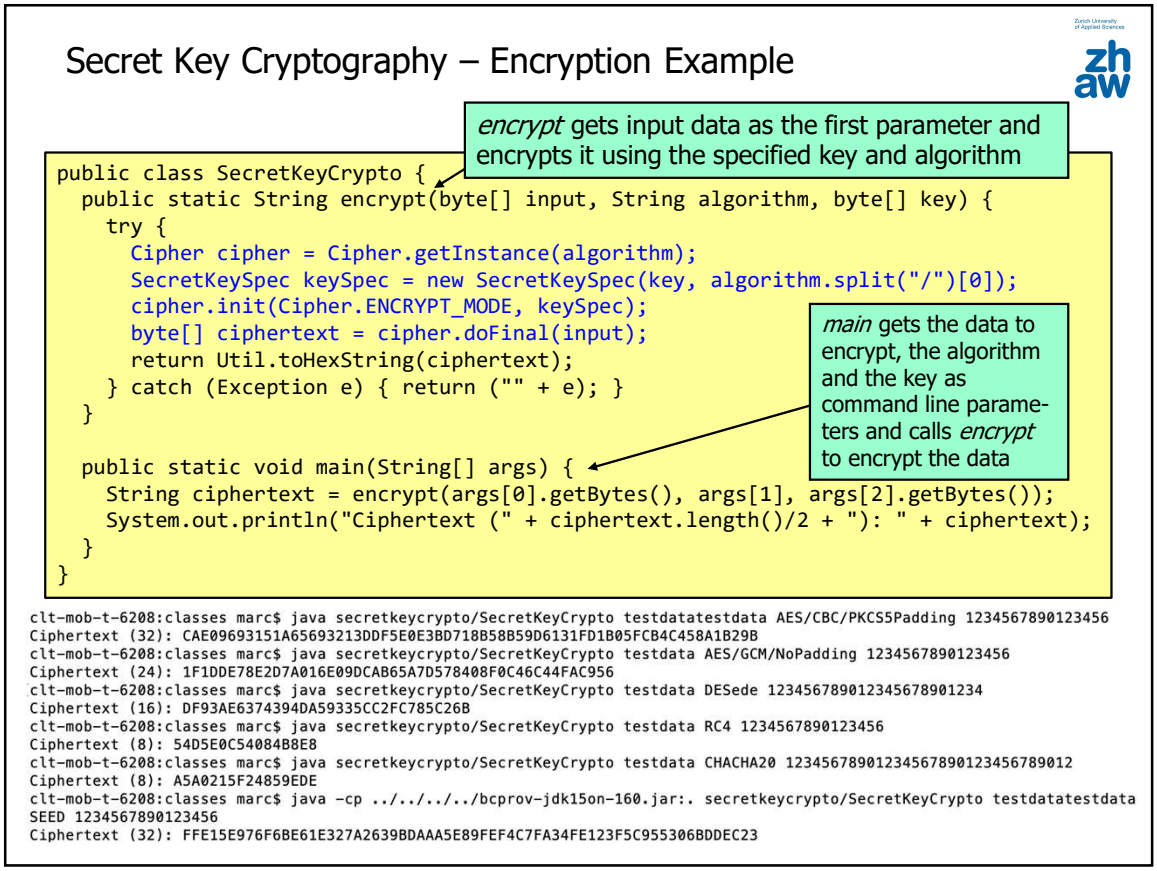
```
Cipher cipher = ...; // Create and init Cipher object for encryption
byte[] output1 = cipher.update("Software securit".getBytes());
byte[] output2 = cipher.update("y is very import".getBytes());
byte[] output3 = cipher.doFinal("ant, trust me!".getBytes());
byte[] output = ...; // Concat output1, output2 and output3
```

```
Cipher cipher = ...; // Create and init Cipher object for encryption
byte[] output = cipher.doFinal("Software security is very
                               important, trust me!".getBytes());
```


doFinal

One could argue that with stream ciphers, *doFinal* is not needed as there is nothing to pad and no padding to be removed. That's basically correct, but it's not guaranteed that the *update* method always processes all bytes that are received as input immediately. For instance, depending on the implementation, it may be that (e.g., for performance optimization reasons) the method processes the received input only if a certain amount of data is available «in the *Cipher*» object. Therefore, only the call to *doFinal* guarantees that all data that may still be available in the *Cipher* object is processed.

Note that *doFinal* not only concludes an encryption / decryption, but also resets the *Cipher* object to its initial state. So, if the same data is now encrypted or decrypted again, then the same output will result. Using the same *Cipher* object again can be highly dangerous, as it will reuse the same IV (violates the rule «never use the same IV twice as long as the key remains unchanged!») and as it will create the same keystream in case of a stream cipher (violates the rule «never use the same keystream twice!»).



Outputs

- *AES/CBC/PKCS5Padding* uses 16 bytes plaintext and produces 32 bytes ciphertext. This is because if the plaintext is a multiple of the block length, a full plaintext block of padding date is used and encrypted.
- *AES/GCM/NoPadding* produces a 24-byte output. This is because in addition to the ciphertext (8 bytes, no padding here), there's an 16-byte authentication tag.
- *DESede* uses 8 bytes plaintext and produces 16 bytes ciphertext. This is because if the plaintext is a multiple of the block length (8 bytes with DES / 3DES), a full plaintext block of padding date is used and encrypted. Note that the key length of 3DES is 192 bits (actually 168 bits as the last bit of every byte is ignored).
- *RC4* uses 8 bytes plaintext and delivers 8 Bytes of ciphertext, because with stream ciphers, there's no padding.
- *CHACHA20* uses 8 bytes plaintext and delivers 8 Bytes of ciphertext, because with stream ciphers, there's no padding.
- *SEED* is a less frequently used block cipher. It is not supported by standard Java, but it is supported by by the Bouncy Castle CSP, which is why the corresponding library is included wit the *-cp* option.

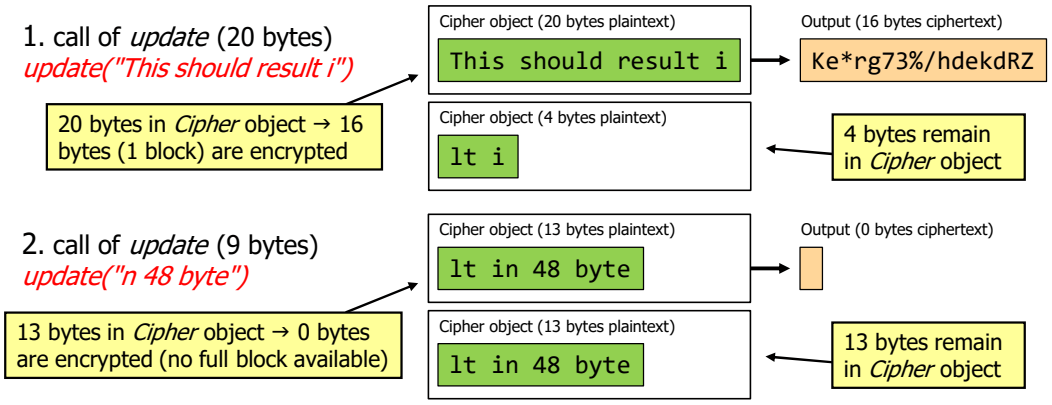
Decryption

To decrypt a ciphertext, simply initialize the *Cipher* object in *DECRYPT_MODE* and use the ciphertext in the *doFinal* method. This delivers the original plaintext. The method below illustrates this (assuming the original plaintext was a string):

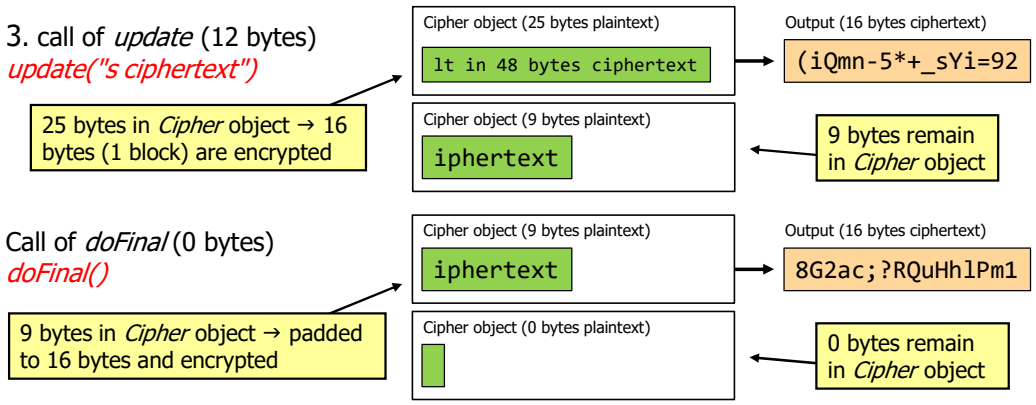
```
public static String decrypt(byte[] input, byte[] key,
                             String algorithm) {
  try {
    Cipher cipher = Cipher.getInstance(algorithm);
    SecretKeySpec keySpec = new SecretKeySpec(key,
                                algorithm.split("/")[0]);
    cipher.init(Cipher.DECRYPT_MODE, keySpec);
    byte[] plaintext = cipher.doFinal(input);
    return new String(plaintext);
  } catch (Exception e) {
    return ("" + e);
  }
}
```

Secret Key Cryptography – Methods *update* and *doFinal* (1)

- To truly understand the behavior of the methods *update* and *doFinal*, they are analyzed in more detail
- Example: Encryption of the 41 bytes long plaintext *This should result in 48 bytes ciphertext* with AES (block size: 16 bytes)



Secret Key Cryptography – Methods *update* and *doFinal* (2)



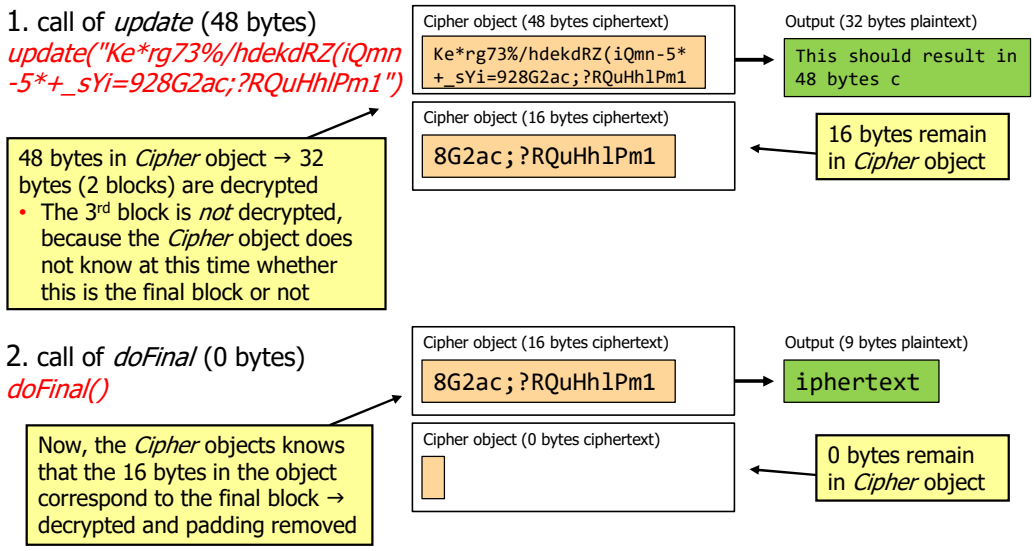
- This shows that *update* is quite «smart» in the sense that it only encrypts the data that can be safely encrypted (e.g., only full blocks)
  - This implies that it does not matter what plaintext chunks are used in *update* as the method makes sure encryption works in every case
- Make sure to call *doFinal* once at the end, so that all remaining data is encrypted and that padding is correctly done

Stream Ciphers

With stream ciphers, *update* always encrypts all bytes, as stream ciphers are bit-oriented and not block-oriented.

Secret Key Cryptography – Methods *update* and *doFinal* (3)

- Example continued: Decryption of the 48 bytes long ciphertext *Ke*rg73%/hdekdRZ(iQmn-5*+_sYi=928G2ac;?RQuHhlPm1* with AES



update during Decryption

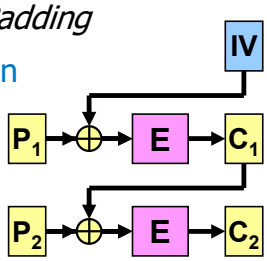
This shows that *update* is also «smart» during decryption. It only decrypts full blocks and it only decrypts blocks if it «can be sure» that an additional block is following, because the final block must be decrypted with *doFinal* to make sure that padding is correctly removed after decryption.

Secret Key Cryptography – Important Remarks

Besides using random key material and using *update* and *doFinal* correctly, secret key cryptography has several additional pitfalls:

- Make sure not to use ECB mode (why?) – which can easily happen as using the cipher name *AES* corresponds to *AES/ECB/PKCS5Padding*
- Don't use secret key cryptography without integrity protection (why?) – so always combine it with a MAC or use a cipher mode with integrated integrity protection such as *AES/GCM/NoPadding*
- Some cipher modes (e.g., CBC) require an initialization vector (IV) → 3rd parameter of *Cipher.init* method

```
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

// Create an initialization vector (IV) with 128 random bits
byte[] iv = new byte[16];
SecureRandom random = new SecureRandom();
random.nextBytes(iv);

// When initializing cipher, also use the IV
cipher.init(mode, key, new IvParameterSpec(iv));
```



This will be discussed in detail in the security lab!

Public Key Cryptography (1)

- Java can also perform public key operations such as digital signatures, encryption and Diffie-Hellman key exchange
  - As an example, we consider here public key encryption and signatures
- Generating a public key pair works similar as generating a secret key, but uses different classes
- The following code serves to generate an RSA key pair:

```
// Create a KeyPairGenerator for RSA keys
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");

// Initialize the generator with the key size (here: 2048 bits)
keyPairGen.initialize(2048);

// Generate the key pair and extract public and private keys
KeyPair keyPair = keyPairGen.generateKeyPair();
PublicKey publicKey = keyPair.getPublic();
PrivateKey privateKey = keyPair.getPrivate();
```

  - Other public key pairs are also supported, e.g., DSA, ECC

Public Key Cryptography (2)

- To encrypt / decrypt data, the class *Cipher* is used (just like with secret key cryptography):

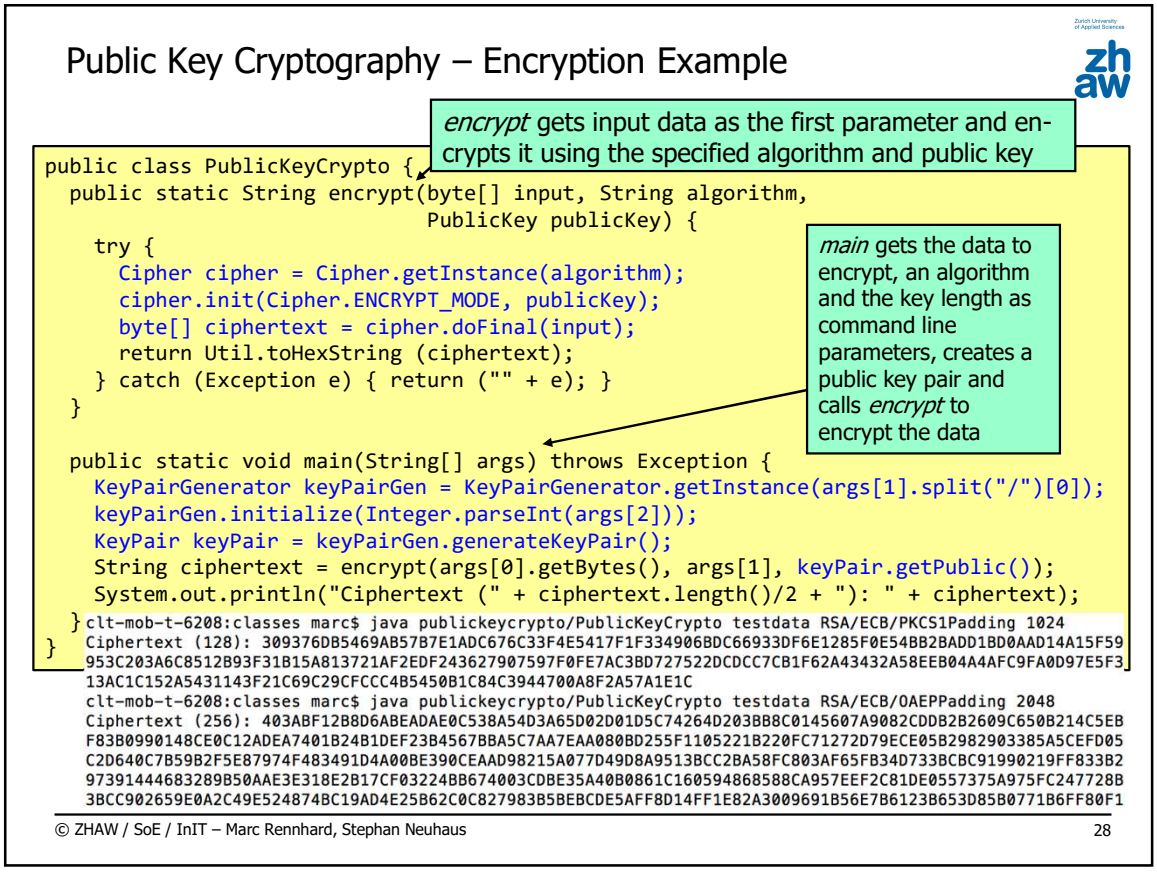
```
// Creating a Cipher object for RSA encryption / decryption
Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPPadding");

// For encryption / decryption, initialize the cipher with the
// desired mode and the correct key
cipher.init(Cipher.ENCRYPT_MODE, publicKey);   // encryption
cipher.init(Cipher.DECRYPT_MODE, privateKey);  // decryption

// To encrypt / decrypt, use methods update and doFinal
cipher.update(...);    cipher.doFinal(...);
```

- Just like with secret key crypto, the parameter of *Cipher.getInstance()* specifies the cipher algorithm, cipher mode and padding scheme, e.g.:
  - *RSA/ECB/OAEPPadding*: RSA with OAEPPadding (≡ PKCS #1 v2 padding)
  - *RSA/ECB/PKCS1Padding*: PKCS #1 v1.5 padding
  - *RSA*: Defaults to *RSA/ECB/PKCS1Padding*
  - It's recommended to use OAEPPadding for security reasons

RSA ECB

Note that with RSA as provided by Java, the amount of data to be encrypted (with *doFinal*) is limited by the amount of data that can be processed in a single RSA operation. E.g., with a 2'048-bit modulus, no more than 256 bytes can be encrypted. Therefore, the mode ECB in the full algorithm name is somewhat pointless as «only one block» can be encrypted. Nevertheless, it's used to indicate that – if multiple RSA encryptions are done – each RSA operation is independent of the others (no chaining between blocks). Note that when doing two RSA encryptions with the same input (e.g., call *doFinal* twice with the same data), this results in two different outputs as the used padding scheme always introduces some randomness between the mapping of plaintext to ciphertext. This means that the typical ECB problems we have with secret key ciphers (where the same plaintext blocks result in the same ciphertext blocks) do not exist with RSA encryption if a good padding scheme is used.

Note that in practice, RSA encryption is usually only used to encrypt a small piece of data, e.g., a secret key in combination with hybrid encryption or a secret key during establishment of a secure communication protocol. This means that typically, performing a single RSA operation «is often enough» in many practical scenarios.



Outputs

The ciphertext corresponds to one RSA block, the length of which corresponds to the RSA key length. In the examples above, this is 128 bytes (1024 bits) and 256 bytes (2048 bytes).

Decryption

To decrypt a ciphertext, simply initialize the *Cipher* object in *DECRYPT_MODE* with the private key and use the ciphertext in the *doFinal* method. This delivers the original plaintext. The method below illustrates this (assuming the original plaintext was a string):

```
public static String decrypt (byte[] input, String algorithm,
                              PrivateKey privateKey) {
  try {
    Cipher cipher = Cipher.getInstance(algorithm);
    cipher.init(Cipher.DECRYPT_MODE, privateKey);
    byte[] plaintext = cipher.doFinal(input);
    return new String(plaintext);
  } catch (Exception e) {
    return ("" + e);
  }
}
```

Public Key Cryptography (3)

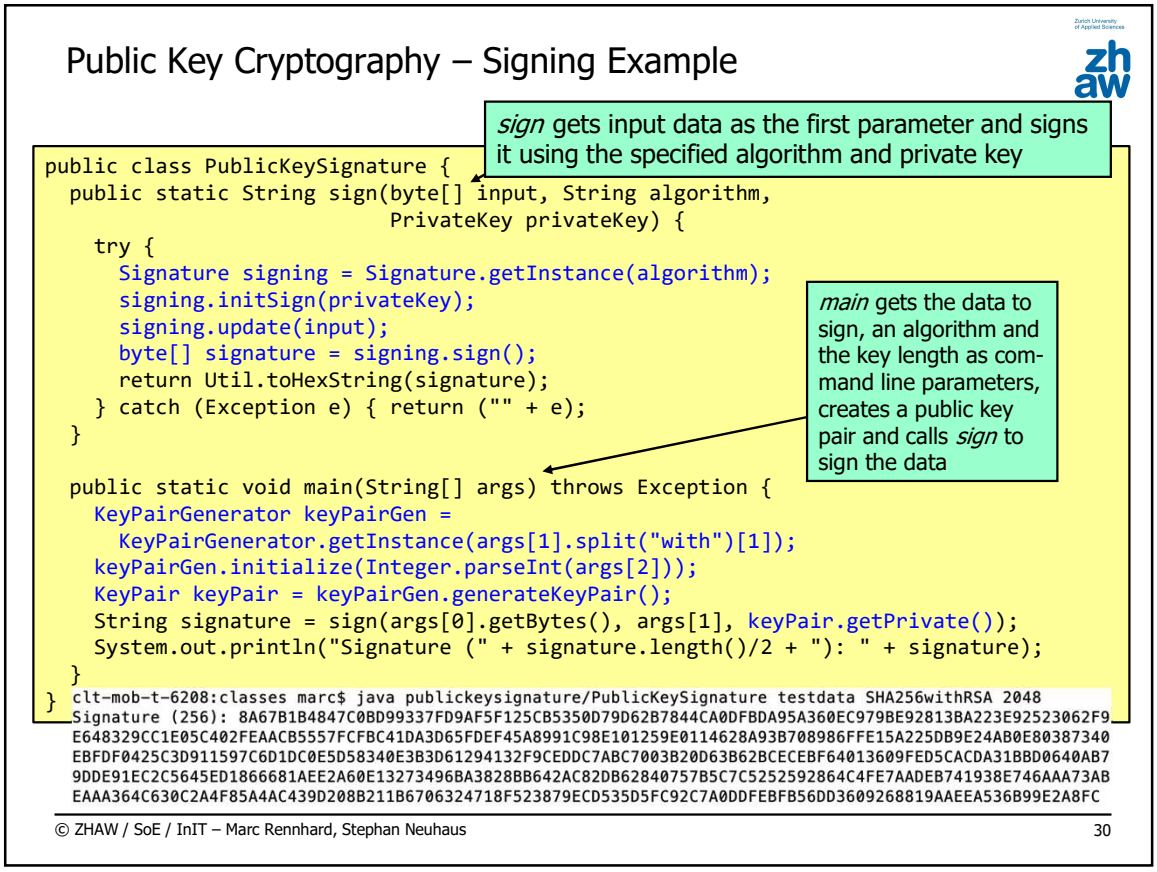
- To sign data and verify signatures, the class *Signature* is used:

```
// Creating a Signature object to sign / verify signatures based on
// SHA2-512 and RSA
Signature signing = Signature.getInstance("SHA512withRSA");

// For signing / verifying, initialize the Signature object with
// the correct key
signing.initSign(privateKey);    // signing
signing.initVerify(publicKey);   // verifying

// To sign, use methods update (once/multiple times) and sign
signing.update(data-to-sign);
byte[] signature = signing.sign();

// To verify, use methods update (once/multiple times) and verify
signing.update(data-that-has-been-signed);
boolean signatureValid = signing.verify(signature);
```

- Several algorithms are supported, e.g.:
  - *SHA512withRSA*, *SHA256withDSA*, *SHA3-512withECDSA*,...
  - The signature format depends on the algorithm (e.g., PKCS #1 with RSA)

PKCS #1 Padding Schemes for RSA Signatures

In the case of signing (unlike encryption), also the older PKCS #1 padding schemes are secure (e.g., v1.5 is fine)



Outputs

A signature corresponds to one block of the used public key algorithm. In the case of RSA, the size of this block corresponds to the RSA key length. In the example above, this is 256 bytes (2048 bits).

Verifying the Signature

To verify a signature, initialize the *Signature* object with *initVerify* by using the public key, use the input data in the *update* method, and then call the *verify* method by passing the signature as a parameter. This delivers *true* or *false*, depending on whether the signature is correct or not. The method below illustrates:

```
public static boolean verify(byte[] input, byte[] signature,
                             PublicKey publicKey, String algorithm) {
  try {
    Signature signing = Signature.getInstance(algorithm);
    signing.initVerify(publicKey);
    signing.update(input);
    boolean signatureValid = signing.verify(signature);
    return signatureValid;
  } catch (Exception e) {
    return false;
  }
}
```

Java Secure Sockets Extension (JSSE)

Java Secure Socket Extension

- Java Secure Socket Extension (JSSE) provides support for the Secure Sockets Layer / Transport Layer Security (SSL/TLS) protocols
  - Implements SSL 3.0 - TLS 1.3, SSL 3.0 disabled per default
- Provides all important features of TLS
  - Support of server- and client-side certificates (via key- and truststores)
  - Supports session resumption which allows to use TLS efficiently
  - Supports OCSP and OCSP stapling to check the status of certificates
  - Allows to customize several settings, e.g., the supported SSL/TLS versions and cipher suites
  - ...
- Actual cryptographic algorithms are provided by JCA
  - JSSE therefore supports the algorithms that are offered by JCA

JSSE

Official guide, see *https://docs.oracle.com/javase/9/security/java-secure-socket-extension-jsse-reference-guide.htm* for the official Java documentation.

TLS

Note that whenever TLS is used on the following slides, this includes the entire SSL / TLS protocol family. As we are typically using only TLS today (and no longer the insecure SSL versions), it's reasonable to use the term TLS instead of the «old» term SSL.

Non-Secure Socket Communication

- First, we look at non-secure socket communication in Java, as changing it to secure communication only requires a few steps

- For this, we are using an Echo application as an example that works as follows:
  - The server listens on port 9999 for incoming TCP connections
  - The client connects to the server
  - Anything that is entered on stdin on the client is sent to the server
  - The server displays (echoes) the received data on stdout

- It's «very basic»:
  - No multithreading
  - The server can just handle one client and exits when the client terminates the connection

Non-Secure Echo Server

```
public class EchoServer {
  public static void main(String[] args) {
    try {

      // Create a ServerSocket that accepts connections on port 9999
      ServerSocket ss = new ServerSocket(9999);

      // Start listening for incoming connections and accept connection
      // requests (blocking); when a client connects, a Socket object is
      // returned to communicate with the client
      Socket socket = ss.accept();

      // Echo anything that is received to stdout
      Scanner reader = new Scanner(socket.getInputStream());
      String string = null;
      while ((string = reader.nextLine()) != null) {
        System.out.println(string);
        System.out.flush();
      }
    } catch (NoSuchElementException exception) {
      // OK, client disconnected
    } catch (Exception exception) {
      exception.printStackTrace();
    }
  }
}
```


Basic approach to implement Server-side

- Create a *ServerSocket* object and set the port on which connections should be accepted.
- The *accept* method is called to wait for connection requests by clients.
- When a connection has been established, the method returns a *Socket* object that can be used to communicate with the client.
- The methods *getInputStream* and *getOutputStream* of the *Socket* object return the stream objects to send and receive data.

Non-Secure Echo Client

```
public class EchoClient {
  public static void main(String [] args) {
    try {
      // Create a Socket that connects to localhost:9999, which can be
      // used to communicate with the server
      Socket socket = new Socket("localhost", 9999);

      // Read lines from stdin and send them to the server
      Scanner reader = new Scanner(System.in);
      OutputStream outputstream = socket.getOutputStream();
      OutputStreamWriter outputstreamwriter =
          new OutputStreamWriter(outputstream);
      BufferedWriter bufferedwriter =
          new BufferedWriter(outputstreamwriter);
      String string = null;
      while ((string = reader.nextLine()) != null) {
        bufferedwriter.write(string + '\n');
        bufferedwriter.flush();
      }
    } catch (Exception exception) {
      exception.printStackTrace();
    }
  }
}
```


Basic approach to implement Client-side

- Create a *Socket* object and specify the host name / IP address and port of the server to which to connect.
- This directly establishes the connection to the server.
- The methods *getInputStream* and *getOutputStream* of the *Socket* object return the stream objects to send and receive data.

A Secure Echo Application – Server Side

- Extending this example to a secure application that uses TLS requires only little adaptation on the server side:
  - Use an *SSLServerSocket* object instead of a *ServerSocket* object, which is created via a *SSLServerSocketFactory*
  - *accept* returns an *SSLSocket* object, which can be used to communicate with the client just like the *Socket* object before

```
public class EchoServer {
  public static void main(String[] args) {
    try {
      // Create the SSLServerSocketFactory with default settings
      SSLServerSocketFactory sslSSF =
          (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();

      // Create an SSLServerSocket that accepts connections on port 9999
      SSLServerSocket ss =
          (SSLServerSocket)sslSSF.createServerSocket(9999);

      // Start listening for incoming connections and accept connection
      // requests (blocking)
      SSLSocket socket = (SSLSocket)ss.accept();

     // Echo anything that is received to stdout
     // No changes from here on...
```


SSLServerSocketFactory.getDefault()

SSLServerSocketFactory.getDefault() creates an *SSLServerSocketFactory* object, but the return type of the method is *ServerSocketFactory*. To assign it to a variable of type *SSLServerSocketFactory*, it must be explicitly casted. Similarly, this must also be done with *SSLServerSocket* and *SSLSocket*.

A Secure Echo Application – Client Side

- On the client side, some small changes must be done as well:
  - Create an *SSLSocketFactory*, which can be used to create an *SSLSocket* object that establishes a connection to the server
  - Once the connection has been established, the *SSLSocket* object can be used to communicate with the server just like the *Socket* object before

```
public class EchoClient {
  public static void main(String[] args) {
    try {

      // Create the SSLSocketFactory with default settings
      SSLSocketFactory sslSF =
          (SSLSocketFactory)SSLSocketFactory.getDefault();

      // Creates an SSLSocket that connects to localhost:9999
      SSLSocket socket = (SSLSocket)sslSF.createSocket("localhost", 9999);

      // Read lines from stdin and send them to the server
      // No changes from here on...
```

A Secure Echo Application – Keystore and Truststore

- Since TLS uses public key pairs and certificates, we have to create them before the example can be run
  - Per default, we have only server-side authentication when TLS is used in Java → we need a key pair and a corresponding certificate for the server
  - And the client has to trust the certificate (directly or via a trusted root certificate) as otherwise, an exception is thrown during the TLS handshake

- To store this information, Java uses keystores and truststores
  - keystores contain both private keys and corresponding certificates
  - truststores contain trusted certificates (but no private keys)
  - Both are created / manipulated using the *keytool* command-line tool
  - The party that authenticates itself (here the server) needs a keystore
  - The party that authenticates the other endpoint (client) needs a truststore
  - The standard truststore shipped with Java (*$JAVA_HOME/lib/security/cacerts*) contains root certificates of official certification authorities (CAs) → certificates issued by these CAs are always trusted

keytool

Java includes the *keytool*, a command-line tool to generate and manage keys and certificates. This tool can be used to administrate keystores and truststores. As keystores and truststores support the standard PKCS #12 format, this could also be done with other tools such as *openssl*.

A Secure Echo Application – Key Pair and Certificate Creation

- Generate the key pair and the self-signed certificate for the server and store it in a keystore with name *ks_server* and format *PKCS #12*

```
keytool -genkeypair -keyalg rsa -keystore ks_server -storetype PKCS12
-alias localhost
```

- The certificate follows the X.509 standard and the naming uses the X.500 naming scheme, so corresponding values have to be entered:

```
Enter keystore password:  password
Re-enter new password:  password
What is your first and last name?
  [Unknown]:  localhost
What is the name of your organizational unit?
  [Unknown]:  SWS1
What is the name of your organization?
  [Unknown]:  ZHAW
What is the name of your City or Locality?
  [Unknown]:  Winterthur
What is the name of your State or Province?
  [Unknown]:  ZH
What is the two-letter country code for this unit?
  [Unknown]:  CH
Is CN=localhost, OU=InIT, O=ZHAW, L=Winterthur, ST=ZH, C=CH correct?
  [no]:  yes
```


alias

The alias is used to identify the key pair / certificate, as a keystore may contain several of them.

keytool

Refer to *https://docs.oracle.com/javase/9/tools/keytool.htm* for a detailed explanation of *keytool* and all command-line tool options. Per default, DSA keys with a length of 2048 bits are generated and SHA256withDSA is used for signatures. With *-keyalg RSA*, usage of RSA keys can be enforced.

Common Name

If you are familiar with certificate handling when browsing the Web using SSL/TLS, you know that browsers display a warning when the Common Name in the server certificate does not match the hostname you entered in the address bar (e.g., https://**www.zhaw.ch**/foo/bar). When using JSSE, this check is not done automatically, so no matter what you enter as a reply to the "What is your first and last name" question (which results in the Common Name entry in the certificate), running the example will not result in any complaint about a mismatch in the name. All JSSE checks is that the certificate can be trusted in the sense that it is included in the client's truststore or it is signed by a certificate in the truststore. Any further checks of the certificate have to be done by the programmer by accessing the certificate received from the peer and analyzing its content.

Just to give you a start: This can be done by using the *getSession* method of the *SSLSocket* object, which returns a *SSLSession* object. Then, the *getPeerCertificates* method of the *SSLSession* object can be used to access the certificate. For details, refer to the official Java security developer's guide.

Using Certificates from other Sources?

While *keytool* can be used to create certificates, it's not really powerful. If you want to set specific certificate options, using, e.g., *openssl* or any other certificate management software is usually the better option. *keytool* is then still used to import the certificates and keys into keystores and truststores. You can, of course, also import «real» certificates and keys issued by certification authorities.

However, if you just need some keys and certificates to do some tests, *keytool* is well suited also for certificate and key creation.

A Secure Echo Application – Export and Show Certificate

- Export the certificate

```
keytool -exportcert -keystore ks_server -alias localhost
-file server.cer
```

- Display the certificate (just to show the functionality):

```
keytool -printcert -file server.cer
```

```
Owner: CN=localhost, OU=SWS1, O=ZHAW, L=Winterthur, ST=ZH, C=CH
Issuer: CN=localhost, OU=SWS1, O=ZHAW, L=Winterthur, ST=ZH, C=CH
Serial number: 613dadd5
Valid from: Mon Jan 01 09:59:33 CET 2021 until: Sun Apr 01 10:59:33 CEST 2021
Certificate fingerprints:
          SHA1: 70:90:93:DE:80:F6:1A:3E:10:21:44:E9:02:6A:D4:06:6A:37:1D:FD
          SHA256: 19:CA:98:82:BD:C6:1F:AB:A5:BA:BF:11:5C:24:67:5C:1D:F4:19:F7:
                  CF:7F:19:F4:F3:49:D9:83:17:A8:85:9E
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3
```

A Secure Echo Application – Truststore Creation

- Create a truststore for the client with name *ts_client* and format *PKCS #12* and import the certificate of the server

```
keytool -importcert -keystore ts_client -storetype PKCS12
-file server.cer -alias localhost
```

```
Enter keystore password:  password
Re-enter new password:  password
Owner: CN=localhost, OU=SWS1, O=ZHAW, L=Winterthur, ST=ZH, C=CH
Issuer: CN=localhost, OU=SWS1, O=ZHAW, L=Winterthur, ST=ZH, C=CH
Serial number: 613dadd5
Valid from: Mon Jan 01 09:59:33 CET 2021 until: Sun Apr 01 10:59:33 CEST 2021
Certificate fingerprints:
         SHA1: 70:90:93:DE:80:F6:1A:3E:10:21:44:E9:02:6A:D4:06:6A:37:1D:FD
         SHA256: 19:CA:98:82:BD:C6:1F:AB:A5:BA:BF:11:5C:24:67:5C:1D:F4:19:F7:
                 CF:7F:19:F4:F3:49:D9:83:17:A8:85:9E
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 3
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

- Sidenote: This example simply uses a self-signed certificate for the server, but certificate hierarchies are also possible (see notes below)

Certificate Hierarchies

In the example used above, we simply use a self-signed certificate. However, keystores can also be used with certificate hierarchies, i.e., a root certificate that is used to sign further certificates, which in turn are used for authentication. The following example is from *Cay S. Horstmann and Gary Cornell, Core Java Volume 2. Advanced Features*:

- Generate a root certificate

```
keytool -genkeypair -keystore acmesoft.certs -alias acmeroot
```

- Export the certificate

```
keytool -exportcert -keystore acmesoft.certs -alias acmeroot -file acmeroot.cer
```

- Add the root certificate to the keystores of the employees, e.g., Cindy:

```
keytool -importcert -keystore cindy.certs -alias acmeroot -file acmeroot.cer
```

Note that when importing the certificate, she is asked whether she trusts it.

- Have Alice generate a key pair and export the certificate:

```
keytool -genkeypair -keystore alice.certs -alias alice
keytool -exportcert -keystore alice.certs -alias alice -file alice.cer
```

- Sign Alice's certificate using the root certificate. This cannot be done with *keytool*, but there's a Java program available on the book's resources pages to do so:

```
java CertificateSigner -keystore acmesoft.certs -alias acmeroot -infile
alice.cer -outfile alice_signedby_acmeroot.cer
```

- Have Cindy import the signed certificate into her keystore. This time – as she already trusts the root certificate, she is not asked to trust Alice's certificate

```
keytool -importcert -keystore cindy.certs -file alice.cer -alias alice
```

Of course, generating the key pairs, certificates and signatures can also be done with another tool such as *openssl*, after which you can install the keys and certificates into keystores with *keytool*.

A Secure Echo Application – Running the Example

- To run the server, specify the keystore and password:

```
$ java -Djavax.net.ssl.keyStore=ks_server
-Djavax.net.ssl.keyStorePassword=password EchoServer
```

- To run the client, specify the truststore and password:

```
$ java -Djavax.net.ssl.trustStore=ts_client
-Djavax.net.ssl.trustStorePassword=password EchoClient
```

- Entering a line in the client....

```
rema:bin marc$ java -Djavax.net.ssl.trustStore=ts_client -Djavax.net.ssl.trustSt
orePassword=password EchoClient
You are the best of the best of the best!
```

- ...echoes it at the server side

```
rema:bin marc$ java -Djavax.net.ssl.keyStore=ks_server -Djavax.net.ssl.keyStoreP
assword=password EchoServer
You are the best of the best of the best!
```


JSSE Debugging

To enable debugging, use the Java option *-Djavax.net.debug=ssl*.

A Secure Echo Application – Analysis

- The Wireshark trace shows that TLS 1.3 is used – which is good!

6	4.898325	127.0.0.1	127.0.0.1	TCP	64990 → distinct(9999) [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=
7	4.898375	127.0.0.1	127.0.0.1	TCP	distinct(9999) → 64990 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344
8	4.898382	127.0.0.1	127.0.0.1	TCP	64990 → distinct(9999) [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=25359298
9	4.898391	127.0.0.1	127.0.0.1	TCP	[TCP Window Update] distinct(9999) → 64990 [ACK] Seq=1 Ack=1 Win=408256
18	17.240357	127.0.0.1	127.0.0.1	TLSv1.3	Client Hello
19	17.240384	127.0.0.1	127.0.0.1	TCP	distinct(9999) → 64990 [ACK] Seq=1 Ack=426 Win=407872 Len=0 TSval=253605
20	17.272182	127.0.0.1	127.0.0.1	TLSv1.3	Server Hello
21	17.272214	127.0.0.1	127.0.0.1	TCP	64990 → distinct(9999) [ACK] Seq=426 Ack=161 Win=408128 Len=0 TSval=2536
22	17.282059	127.0.0.1	127.0.0.1	TLSv1.3	Change Cipher Spec

- The client offers only strong cipher suites

```
Cipher Suites (44 suites)
    Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
    Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
    Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02e)
    Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384 (0xc032)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x009f)
    Cipher Suite: TLS_DHE_DSS_WITH_AES_256_GCM_SHA384 (0x00a3)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
    Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02d)
    Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256 (0xc031)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x009e)
    Cipher Suite: TLS_DHE_DSS_WITH_AES_128_GCM_SHA256 (0x00a2)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA256 (0x003d)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384 (0xc026)
    Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384 (0xc02a)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (0x006b)
    Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA256 (0x006a)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA (0xc005)
    Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA (0xc00f)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
    Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA256 (0x003c)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256 (0xc025)
    Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256 (0xc029)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA256 (0x0067)
    Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA256 (0x0040)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
    Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA (0xc004)
    Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA (0xc00e)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
    Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)
```

A more Elaborate Example

- The previous example demonstrated the easiest way to work with TLS in Java
  - The default configuration is «secure»
- We now modify the secure Echo application to show some additional possibilities you have when working with JSSE
- In particular, we implement the following extensions:
  - We do not specify keystores and truststores via the command-line, but access them directly from within the program
  - We specify the TLS version to be used
  - To do so, we cannot use *SSL(Server)SocketFactory* with default settings, but must create them via an *SSLContext* object
  - We use additional client authentication
  - We specify the cipher suites to be used
  - The server displays some information about the client's certificate

A more Elaborate Example – keystores and truststores

- We need some keystores and truststores:
  - Both the client and the server need a keystore that contains a private key and a corresponding certificate, as both authenticate themselves
  - Both need a truststore to verify the peer's certificate

```
Create a key pair for server and client using RSA keys with a 4096-bit modulus
(default is 2048 bits):

keytool -genkeypair -keystore ks_server -storetype PKCS12 -keyalg rsa
-keysize 4096 -alias server
keytool -genkeypair -keystore ks_client -storetype PKCS12 -keyalg rsa
-keysize 4096 -alias client

Export the certificates:

keytool -export -keystore ks_server -alias server -file cert_server.cer
keytool -export -keystore ks_client -alias client -file cert_client.cer

Import the peer's cerificate in a truststore:

keytool -import -keystore ts_server -storetype PKCS12
-file cert_client.cer -alias client
keytool -import -keystore ts_client -storetype PKCS12
-file cert_server.cer -alias server
```

A more Elaborate Example – Server Side (1)
Load keystore and truststore

```
public class EchoServer {
  public static void main(String[] args) {

    // The password used for keystores, truststores and private keys
    final char[] PASSWORD = "password".toCharArray();

    try {
      // Create a KeyStore object and load it with the keystore data
      KeyStore keystore = KeyStore.getInstance("PKCS12");
      keystore.load(new FileInputStream("ks_server"), PASSWORD);

      // Create a KeyManagerFactory object and initialize it with the
      // keystore; giving it access to private key and certificate
      KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
      kmf.init(keystore, PASSWORD);

      // Create a KeyStore object and load it with the truststore data
      KeyStore truststore = KeyStore.getInstance("PKCS12");
      truststore.load(new FileInputStream("ts_server"), PASSWORD);

      // Create a TrustManagerFactory object and initialize it with the
      // truststore; giving it access to the certificate
      TrustManagerFactory tmf =
          TrustManagerFactory.getInstance("PKIX");
      tmf.init(truststore);
      ...
```


Keystore Password

Obviously, it's not a good idea to hardcode a password into a program, but it's nevertheless done here for simplicity. And "password" is not a really strong password, I guess...

KeyStore keystore = KeyStore.getInstance("PKCS12")

In our case, we are using the PKCS #12 format for keystores and truststores. Therefore, PKCS12 is used as a parameter in the *getInstance* method to get the right type of *Keystore* object.

KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
TrustManagerFactory tmf = TrustManagerFactory.getInstance("PKIX");

PKIX is a Standard that defines X.509 certificates and how certificates and certificates chains are used. As we are using X.509 certificates here, we create *KeyManagerFactory* and *TrustManagerFactory* objects based on this standard, which makes sure that private keys and certificates are handled correctly.

A more Elaborate Example – Server Side (2)
Create a server socket

```
...

// Create an SSLContext object that uses TLS 1.2
SSLContext sslContext = SSLContext.getInstance("TLSv1.2");

// Create KeyManager and TrustManager objects from the previously
// created factories and use them to initialize the SSLContext object,
// which then uses the private key and certificates in our keystore
// and truststore (the third argument can be used to specify a source
// of randomness for key material generation, default is SecureRandom)
sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

// Create an SSLServerSocketFactory from the SSLContext object
SSLServerSocketFactory sslSSF =
    (SSLServerSocketFactory)sslContext.getServerSocketFactory();

// Create an SSLServerSocket that is bound to port 9999
SSLServerSocket ss = (SSLServerSocket)sslSSF.createServerSocket(9999);

// Require client authentication
ss.setNeedClientAuth(true);

...
```


SSLContext.getInstance("TLSv1.2")

The version specifies the highest version which will be supported by *SSLServerSocket* objects that will be created based on this *SSLContext*. With TLSv1.2, the *SSLServerSocket* objects will support TLS 1, TLS 1.1, and TLS 1.2, as these are the TLS versions «from TLS 1.2 downwards» that are enabled in Java. SSL 3 is supported in Java, but not enabled per default.

SSLContext.init()

Note that the *getKeyManagers* and *getTrustManagers* methods return arrays of *KeyManagers/TrustManagers*. The reason is that if the used keystore/truststore contains different types of keys and certificates, then one *KeyManager/TrustManager* per type is returned. In our case, the keystore/truststore only contains X.509 certificates and corresponding private keys and correspondingly, the returned array also only contains one *KeyManager/TrustManager* (of type *X509KeyManager* and *X509TrustManager*).

A more Elaborate Example – Server Side (3)
Set cipher suites and print certificate information

```
...

// The server just accepts one cipher suite
String[] enabledCipherSuites =
    {"TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256"};
ss.setEnabledCipherSuites(enabledCipherSuites);

// Start listening for incoming connections and accept connection
// requests (blocking)
SSLSocket socket = (SSLSocket)ss.accept();

// Get the client's certificate(s)
X509Certificate[] certificates =
    (X509Certificate[])socket.getSession().getPeerCertificates();
System.out.println("Subject: " +
       certificates[0].getSubjectX500Principal());
System.out.println("Validity: " + certificates[0].getNotBefore() +
       " - " + certificates[0].getNotAfter());

// Echo anything that is received to stdout
// No changes from here on...
```


getPeerCertificates()

The method returns an array of X509Certificates. The reason why there are multiple certificates is that the entire certificate chain (if any) is returned. As our example only uses a self-signed certificate and no hierarchy, just one certificate is returned.

A more Elaborate Example – Client Side (1)
Load keystore and truststore

```
public class EchoClient {
  public static void main(String[] args) {

    // The password used for keystores, truststores and private keys
    final char[] PASSWORD = "password".toCharArray();

    try {
      // Create a KeyStore object and load it with the keystore data
      KeyStore keystore = KeyStore.getInstance("PKCS12");
      keystore.load(new FileInputStream("ks_client"), PASSWORD);

      // Create a KeyManagerFactory object and initialize it with the
      // keystore; giving it access to private key and certificate
      KeyManagerFactory kmf = KeyManagerFactory.getInstance("PKIX");
      kmf.init(keystore, PASSWORD);

      // Create a KeyStore object and load it with the truststore data
      KeyStore truststore = KeyStore.getInstance("PKCS12");
      truststore.load(new FileInputStream("ts_client"), PASSWORD);

      // Create a TrustManagerFactory object and initialize it with the
      // truststore; giving it access to the certificate
      TrustManagerFactory tmf =
          TrustManagerFactory.getInstance("PKIX");
      tmf.init(truststore);
      ...
```


Load Keystore and Truststore

The code above is exactly the same as on the server side, except that different files are used.

A more Elaborate Example – Client Side (2)
Create a socket and set cipher suites

```
...
// Create an SSLContext object that uses TLS 1.2
SSLContext sslContext = SSLContext.getInstance("TLSv1.2");

// Create KeyManager and TrustManager objects from the previously
// created factories and initialize the SSLContext object, which then
// uses the private key and certificates in our keystore and truststore
sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

// Create an SSLSocketFactory from the SSLContext
SSLSocketFactory sslSF =
    (SSLSocketFactory)sslContext.getSocketFactory();

// Creates an SSLSocket that connects to localhost:9999
SSLSocket socket = (SSLSocket)sslSF.createSocket("localhost", 9999);

// The client only wants to use the following cipher suites
String[] enabledCipherSuites = "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384",
                               "TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256",
                               "TLS_RSA_WITH_AES_256_CBC_SHA256",
                               "TLS_RSA_WITH_AES_128_CBC_SHA256"};
socket.setEnabledCipherSuites(enabledCipherSuites);

// Read lines from stdin and send them to the server
// No changes from here on...
```

A more Elaborate Example – Running the Example

- No options are necessary to run server and client:

```
$ java EchoServer
```

```
$ java EchoClient
```

- Entering a line in the client....

```
$ java EchoClient
You are the best of the best of the best!
```

- ...results in the following at the server side

```
$ java EchoServer
Subject: CN=Marc Rennhard, OU=InIT, O=SWS1, L=Winterthur, ST=ZH, C=CH
Validity: Thu Mar 01 15:08:32 CET 2021 - Wed May 30 16:08:32 CEST 2021
You are the best of the best of the best!
```

  - As expected, the output contains information from the client's certificate

Performed Certificate Checks

Besides checking whether the trust chain can be successfully built from the certificate received from the peer to a trusted certificate in the truststore (this of course includes checking the correctness of the signatures), virtually no further security checks are performed. It was noted before that with JSSE, the common name is not checked automatically, this has to be done programmatically if desired. The same is true for other checks. For instance, it is not checked whether the current date is within the validity period of all certificates used in the certificate chain. This can be done using the *checkValidity* method of a certificate (*X509Certificate* class). In addition, revocation checks using CRLs or OCSP must also be done programmatically if desired. For more details, refer to the official Java security developer's guide

A more Elaborate Example – Analysis

- As configured, TLS 1.2 and client authentication is used:
  - The server sends a *Certificate Request* message
  - The client sends its certificate and a *Certificate Verify* message

```
 3 2.047307   127.0.0.1   127.0.0.1   TLSv1.2  Client Hello
 4 2.047329   127.0.0.1   127.0.0.1   TCP      distinct(9999) → 52707 [ACK] Seq=1 Ack=217 Win=6376 Len=0 TSval=
 5 2.236326   127.0.0.1   127.0.0.1   TLSv1.2  Server Hello
 6 2.236351   127.0.0.1   127.0.0.1   TCP      52707 → distinct(9999) [ACK] Seq=217 Ack=91 Win=6378 Len=0 TSval
 7 2.236629   127.0.0.1   127.0.0.1   TLSv1.2  Certificate
 8 2.236645   127.0.0.1   127.0.0.1   TCP      52707 → distinct(9999) [ACK] Seq=217 Ack=1487 Win=6356 Len=0 TSv
 9 2.516446   127.0.0.1   127.0.0.1   TLSv1.2  Server Key Exchange
10 2.516472   127.0.0.1   127.0.0.1   TCP      52707 → distinct(9999) [ACK] Seq=217 Ack=2081 Win=6347 Len=0 TSv
11 2.521941   127.0.0.1   127.0.0.1   TLSv1.2  Certificate Request
12 2.521962   127.0.0.1   127.0.0.1   TCP      52707 → distinct(9999) [ACK] Seq=217 Ack=2241 Win=6344 Len=0 TSv
13 2.522125   127.0.0.1   127.0.0.1   TLSv1.2  Server Hello Done
14 2.522134   127.0.0.1   127.0.0.1   TCP      52707 → distinct(9999) [ACK] Seq=217 Ack=2250 Win=6344 Len=0 TSv
15 2.607501   127.0.0.1   127.0.0.1   TLSv1.2  Certificate
16 2.607529   127.0.0.1   127.0.0.1   TCP      distinct(9999) → 52707 [ACK] Seq=2250 Ack=1621 Win=6354 Len=0 TS
17 2.611242   127.0.0.1   127.0.0.1   TLSv1.2  Client Key Exchange
18 2.611264   127.0.0.1   127.0.0.1   TCP      distinct(9999) → 52707 [ACK] Seq=2250 Ack=1696 Win=6353 Len=0 TS
19 2.814261   127.0.0.1   127.0.0.1   TLSv1.2  Certificate Verify
```

- Offered and selected cipher suites are used as configured:

```
Cipher Suites (4 suites)
   Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
   Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
   Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA256 (0x003d)
   Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA256 (0x003c)
```

```
Handshake Protocol: Server Hello
   Handshake Type: Server Hello (2)
   Length: 81
   Version: TLS 1.2 (0x0303)
▶  Random: 547624fb7879118b99058f14919dd387c05dcbf6332da1a6...
   Session ID Length: 32
   Session ID: ed533f3b8271b135c0946dd390c0e6c9fa89b1b67725ff19...
   Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
```

Summary

- JCA provides fundamental cryptographic operations, including
  - Hash functions, random number generation, message authentication codes, secret key cryptography, public key cryptography, Diffie-Hellman key exchange
- JSSE allows to secure networked applications using TLS
  - Provides all important features of TLS and allows configuring various options such as using client-authentication and supported cipher suites
- Use these components when you need the corresponding functionality
  - Don't invent or implement your own cryptographic algorithms or secure communication protocols!
  - But: it's still necessary to use the libraries with care (make sure to use secure algorithms, key lengths, cipher modes, cipher suites,...)

Appendix

AESTest – Example

- As a complete example including secret key generation, encryption and decryption, we implement a command line program *AESTest*
  - The purpose of this program is to demonstrate how the methods *update* and *doFinal* are used in detail
- The program can be used as follows:
  - Generate a secret key and store it in the file *secretKey*

```
java AESTest -genkey secretKey
```

  - Encrypt the content in the file *plaintext* using the key in the file *secretKey* and store the ciphertext in file *ciphertext*

```
java AESTest -encrypt plaintext ciphertext secretKey
```

  - Decrypt the content in the file *ciphertext* using the key in the file *secretKey* and store the decrypted text in the file *decrypted*

```
java AESTest -decrypt ciphertext decrypted secretKey
```


AESTest Example

The example is based on the AESTest example in *Cay S. Horstmann and Gary Cornell, Core Java Volume 2. Advanced Feature*

You can get the full source code at *http://horstmann.com/corejava/index.html*

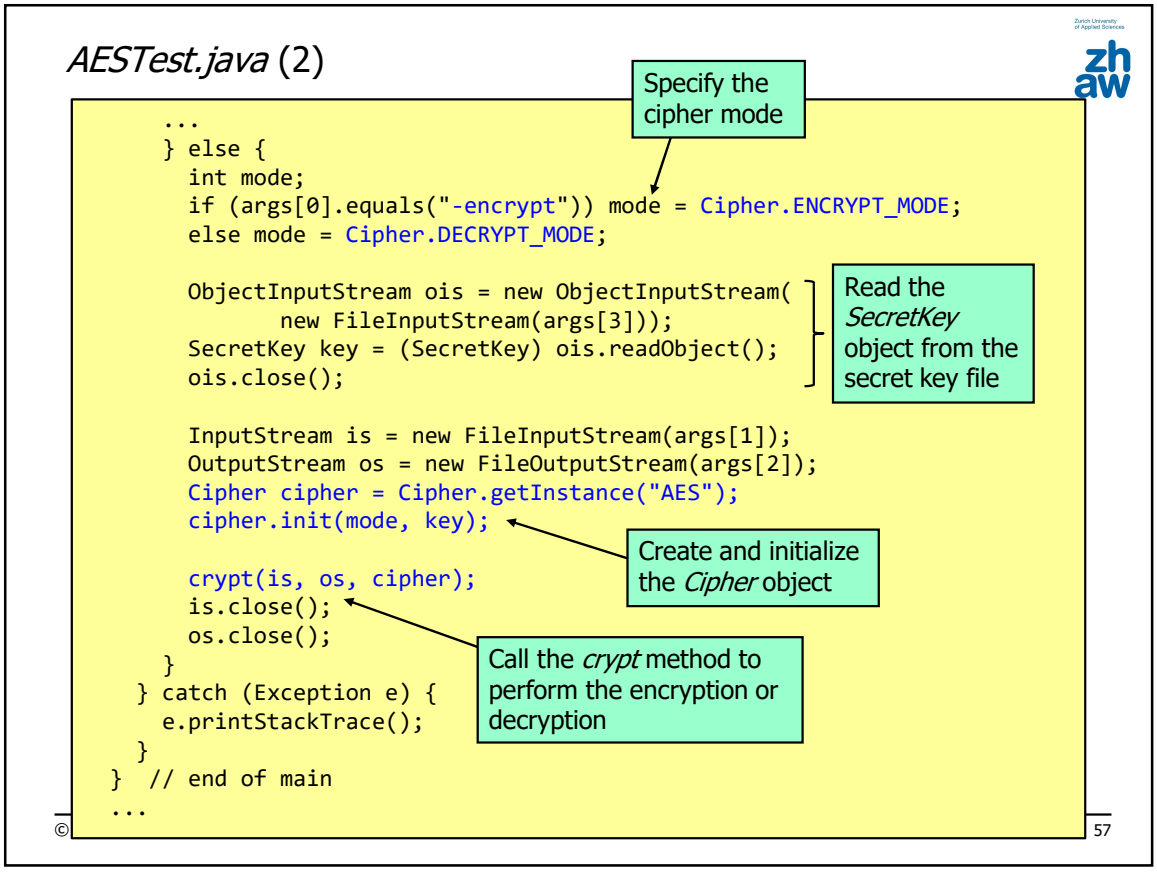
AESTest.java (1)

Note: This is not a secure example as encryption should not be used without integrity protection (e.g., a MAC or using a cipher mode such as GCM). This example merely serves to illustrate a complete raw encryption/decryption example.

```
public class AESTest {

  private static final int KEYSIZE = 128;
  private static final int INPUTSIZE = 1000;

  public static void main(String[] args) {

    try {

      if (args[0].equals("-genkey")) {

        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(KEYSIZE);
        SecretKey key = keyGen.generateKey();

        ObjectOutputStream oos = new ObjectOutputStream(
               new FileOutputStream(args[1]));
        oos.writeObject(key);
        oos.close();
      } else {
        ...
```

Generate a random 128-bit AES key

Write the entire *SecretKey* object to the secret key file (using serialization)



ECB vs. CBC

You probably have heard before that CBC (Cipher Block Chaining) mode is more secure than ECB (Electronic Codebook) mode. You are totally right. Nevertheless, we use ECB here for simplicity so we don't have to deal with initialization vectors.

AESTest.java (3)

```
...
public static void crypt(InputStream is, OutputStream os,
    Cipher cipher) throws IOException, GeneralSecurityException {

  byte[] input = new byte[INPUTSIZE];
  int inBytes;
  boolean more = true;

  while (more) {
    inBytes = is.read(input);

    if (inBytes > 0) {
      os.write(cipher.update(input, 0, inBytes));
    } else {
      more = false;
    }
  }
  os.write(cipher.doFinal());
 }
}
```

The input file can have any size → we process it in chunks of (at most) 1000 bytes (any other size works as well)

Read the next bytes from *is* into *input* (at most 1000 bytes)

If >0 bytes were read, encrypt / decrypt them with *update* and write the output to *os*

No bytes were read → end of file is reached → exit loop

Complete processing the input with *doFinal* and write the output to *os*

update and *doFinal*

As method *update* is «smart» (as discussed before), the code above will always work, for any file size. And a nice side effect of the way *update* works is that exactly the same code can be used for encryption and decryption, which is what we did in the *crypt* method above. But this also shows that to use the *Cipher* class correctly (and securely!), one really must understand how it works and how *update* and *doFinal* must be used.

Running *AESTest*

- We first generate a secret key and then encrypt two files – *plaintext-short* and *plaintext-long* – and decrypt them again:

```
$ java AESTest -genkey secretKey
$ java AESTest -encrypt plaintext-short ciphertext-short secretKey
$ java AESTest -decrypt ciphertext-short decrypted-short secretKey
```

- Listing of the files:

```
$ ls -al
-rw-r--r--  1 marc  staff    2678 Feb  4 13:23 AESTest.class
-rw-r--r--  1 marc  staff  345616 Feb  4 13:24 ciphertext-long
-rw-r--r--  1 marc  staff      64 Feb  4 13:24 ciphertext-short
-rw-r--r--  1 marc  staff  345604 Feb  4 13:24 decrypted-long
-rw-r--r--  1 marc  staff      51 Feb  4 13:24 decrypted-short
-rw-r--r--  1 marc  staff  345604 Feb  4 13:18 plaintext-long
-rw-r--r--  1 marc  staff      51 Feb  4 13:18 plaintext-short
-rw-r--r--  1 marc  staff     141 Feb  4 13:23 secretKey
```

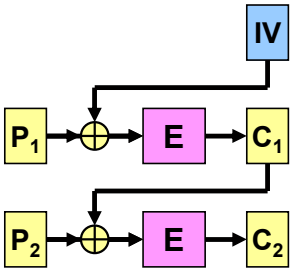
  - *AESTest* works for short and long files, i.e., it works for files of «any» size
  - Due to padding, the ciphertext is always slightly longer than the plaintext and a multiple of the block size (16 byte). E.g., 51 bytes → 64 bytes

ECB Mode is Evil

- Besides not using integrity protection, *AESTest* has an additional security problem: it uses Electronic Codebook mode (ECB)
  - Because the cipher name *AES* corresponds to *AES/ECB/PKCS5Padding*
  - ECB means that plaintext blocks are directly mapped to ciphertext block, which can leak information
- For example, if *AESTest* used this plaintext block (51 bytes, hexdump) where the first and third plaintext blocks are equal...
  - `0000000 30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46`
    `0000010 61 61 62 62 63 63 64 64 65 65 66 66 67 67 68 68`
    `0000020 30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46`
    `0000030 69 69 0a`
- ... then the corresponding ciphertext blocks are also equal, which leaks some information to an observer
  - `0000000 eb b9 d4 f0 39 c9 31 18 6b f8 14 98 f6 d2 38 65`
    `0000010 6b 9d a2 65 6e 63 f7 ea 6e e6 e7 ff bc 89 21 b2`
    `0000020 eb b9 d4 f0 39 c9 31 18 6b f8 14 98 f6 d2 38 65`
    `0000030 fc a1 06 97 8a 33 a8 84 61 f2 18 28 8c be 3d c2`

Using Initialization Vectors / Cipher Block Chaining Mode

- Therefore, never use ECB mode, but (at least) use Cipher Block Chaining (CBC) mode, which prevents the problems of ECB
- When using CBC, an additional parameters is needed to initialize the *Cipher* object → 3rd parameter of *init* method
- Example: *AES/CBC/PKCS5Padding*



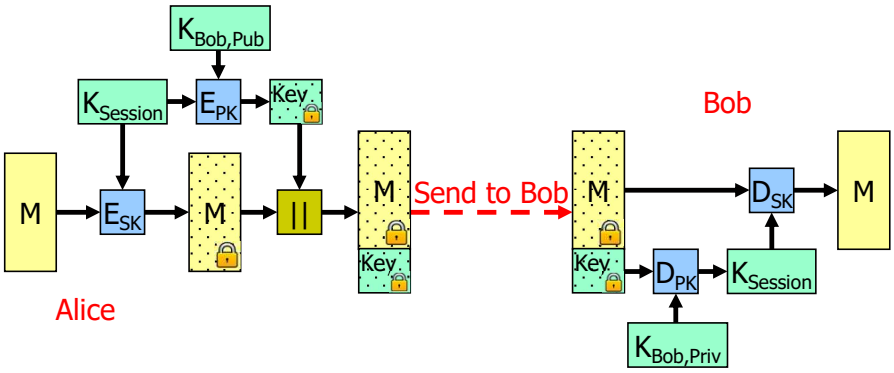

```
// Specify a Cipher in CBC mode
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

// Create an initialization vector (IV) with 128 random bits
byte[] iv = new byte[16];
SecureRandom random = new SecureRandom();
random.nextBytes(iv);

// Create an IvParameterSpec object based on the IV
IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);

// Use the init method with three parameters to pass the
// IvParameterSpec
cipher.init(mode, key, ivParameterSpec);
```


Implicit Parameters

Instead of explicitly specifying the parameters (as shown above), they can also be specified implicitly. In this case, the *Cipher* object is initialized without the third parameter. Once the Cipher object has been created, the parameters can be extracted with *getParameters()*. In general, it's recommended to explicitly specify these parameters as this allows the developer to clearly define them according to his or her requirements.

Hybrid Encryption (1)

- If larger amounts of data should be encrypted with a public key, this shouldn't be done directly, as public key cryptography is relatively slow
- Instead, hybrid encryption should be used, which combines secret and public key cryptography:



- With hybrid encryption, public key encryption is only used to encrypt a short secret key, which typically requires just one public key operation

Hybrid Encryption (2)

- Java supports encrypting a secret key with a public key with two special cipher modes: *Cipher.WRAP_MODE* and *UNWRAP_MODE*

```
// The secret key to be encrypted with the public key
SecretKey secretKey = ...;

// The public key pair used for encryption / decryption
PublicKey publicKey = ...;   PrivateKey privateKey = ...;

// The cipher for RSA encryption / decryption
Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPPadding");

// Encrypting the secret key: Init the cipher with WRAP_MODE and the
// public key and encrypt the secret key with the public key
cipher.init(Cipher.WRAP_MODE, publicKey);
byte[] encryptedKey = cipher.wrap(secretKey);

// Decrypting the secret key: Init the cipher with UNWRAP_MODE and the
// private key and decrypt the secret key with the private key
cipher.init(Cipher.UNWRAP_MODE, privateKey);
Key secretKey = cipher.unwrap(encryptedKey, "AES", Cipher.SECRET_KEY);
```


Cipher Modes

One could also use *ENCRYPT_MODE* instead of *WRAP_MODE*. But using the *WRAP* / *UNWRAP_MODE* is more convenient as the cipher can encrypt *Key* objects and produce *Key* objects when decrypting.

Cipher.unwrap

The *unwrap* method requires 3 parameters:

- The first is the encrypted (wrapped) key (a byte-array).
- The second is a string that specifies the algorithm associated with the encrypted key, e.g., AES or BLOWFISH.
- The third parameter is the type of the wrapped key, which must be one of *SECRET_KEY*, *PRIVATE_KEY*, or *PUBLIC_KEY*.

HybridTest – Example

- As an example for hybrid encryption (based on RSA and AES), we implement a command line program *HybridTest*
- The program can be used as follows:
  - Generate a public key pair and store it in files *publicKey* and *privateKey*

```
java HybridTest -genkey publicKey privateKey
```

  - Encrypt the content in the file *plaintext* using the key in the file *publicKey* and store the ciphertext in file *ciphertext*

```
java HybridTest -encrypt plaintext ciphertext publicKey
```

  - Decrypt the content in the file *ciphertext* using the key in the file *privateKey* and store the decrypted text in the file *decrypted*

```
java HybridTest -decrypt ciphertext decrypted privateKey
```

- The ciphertext file must contain the RSA-encrypted secret key and the AES-encrypted plaintext and uses the following format:

RSATest Example

The example is based on the RSATest example in *Cay S. Horstmann and Gary Cornell, Core Java Volume 2. Advanced Feature*

You can get the full source code at *http://horstmann.com/corejava/index.html*

HybridTest.java (1)

Note: Just like *AESTest* before, this is not a secure example as it does not include integrity protection. This example merely serves to illustrate a complete raw hybrid encryption/decryption example.

```
public class HybridTest {

  private static final int RSA_KEYSIZE = 2048;
  private static final int AES_KEYSIZE = 128;

  public static void main(String[] args) {
    try {
      if (args[0].equals("-genkey")) {

        KeyPairGenerator keyPairGen =
               KeyPairGenerator.getInstance("RSA");
        keyPairGen.initialize(RSA_KEYSIZE);
        KeyPair keyPair = keyPairGen.generateKeyPair();

        ObjectOutputStream oos = new ObjectOutputStream(
               new FileOutputStream(args[1]));
       oos.writeObject(keyPair.getPublic());
       oos.close();
       oos = new ObjectOutputStream(new FileOutputStream(args[2]));
       oos.writeObject(keyPair.getPrivate());
       oos.close();
     } else if ...
```

Generate a 2048-bit RSA key pair

Write the keys to the public / private key files (using serialization)

HybridTest.java (2)

```
...   } else if (args[0].equals("-encrypt")) {
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(AES_KEYSIZE);
        SecretKey secretKey = keyGen.generateKey();

        ObjectInputStream ois = new ObjectInputStream(
               new FileInputStream(args[3]));
        PublicKey publicKey = (PublicKey) ois.readObject();
        ois.close();

        Cipher cipher = Cipher.
               getInstance("RSA/ECB/OAEPPadding");
        cipher.init(Cipher.WRAP_MODE, publicKey);
        byte[] encryptedKey = cipher.wrap(secretKey);
        OutputStream os =
               new FileOutputStream(args[2]);
        os.write(encryptedKey);

        InputStream is = new FileInputStream(args[1]);
        cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        crypt(is, os, cipher);
        is.close();
        os.close();
      } else { ...
```

Generate a 128-bit AES key

Read the *PublicKey* object from the public key file

Encrypt (wrap) the secret key with the public key and write it to the ciphertext file

Encrypt the data in the plaintext file with AES and write it to the ciphertext file (using the same *crypt* method as in *AESTest.java*)

HybridTest.java (3)

```
...   } else {
        InputStream is =
               new FileInputStream(args[1]);
        byte[] encryptedKey = new byte[RSA_KEYSIZE / 8];
        is.read(encryptedKey);

        ObjectInputStream ois = new ObjectInputStream(
               new FileInputStream(args[3]));
        PrivateKey privateKey = (PrivateKey) ois.readObject();
        ois.close();

        Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPPadding");
        cipher.init(Cipher.UNWRAP_MODE, privateKey);
        Key secretKey = cipher.unwrap(encryptedKey, "AES", Cipher.SECRET_KEY);

        OutputStream os = new FileOutputStream(args[2]);
        cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        crypt(is, os, cipher);
        is.close();
        os.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
  }  // end of main
  ...
```

else case: decrypt the ciphertext

Read encrypted secret key from the ciphertext file

Read the *PrivateKey* object from the private key file

Decrypt (unwrap) the secret key with the private key

Decrypt the data in the ciphertext file with AES and write it to the plaintext file (using the same *crypt* method as in *AESTest.java*)

Running *HybridTest.java*

- We first generate a public key pair and then encrypt a file and decrypt it again:

```
$ java HybridTest -genkey publicKey privateKey
$ java HybridTest -encrypt plaintext ciphertext publicKey
$ java HybridTest -decrypt ciphertext decrypted private.key
```

- Listing of the files:

```
$ ls -al
-rw-r--r--  1 marc  staff  4496 Aug 20 08:43 HybridTest.class
-rw-r--r--  1 marc  staff   320 Aug 20 09:20 ciphertext
-rw-r--r--  1 marc  staff    51 Aug 20 09:20 decrypted
-rw-r--r--  1 marc  staff    51 Aug 20 08:22 plaintext
-rw-r--r--  1 marc  staff  1478 Aug 20 09:19 privateKey
-rw-r--r--  1 marc  staff   551 Aug 20 09:19 publicKey
```

  - *plaintext* and *decrypted* have a size of 51 bytes
  - *ciphertext* has a size of 320 bytes → this consists of the RSA-encrypted secret key (256 bytes) and the AES-encrypted ciphertext (64 bytes due to padding to multiple of block size)