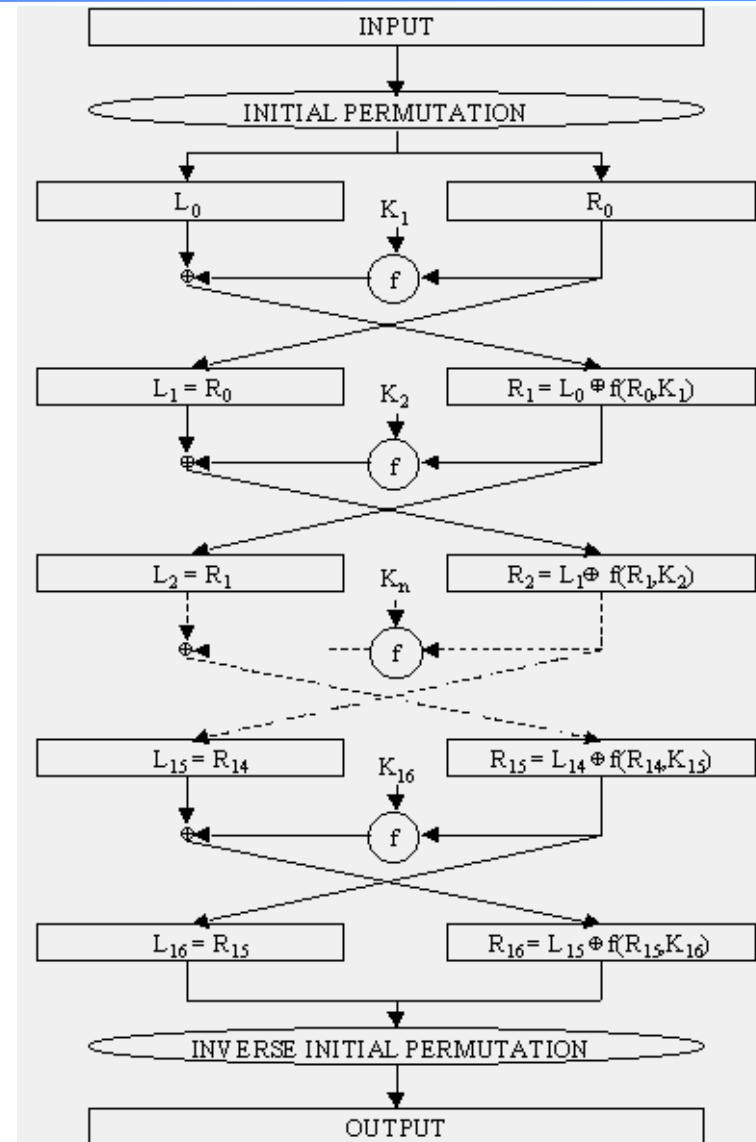

API segurança do Java

Cifras simétricas
(inclui síntese e MACs)

Motivação

- ❖ Algumas primitivas criptográficas que queremos usualmente utilizar:
 - Sínteses seguras;
 - Cifra simétrica;
 - Cifra de chave pública;
 - Assinaturas Digitais;
 - *Message Authentication Codes* (MACs);
 - Canais seguros, etc.
- ❖ Será que programar uma dessas primitivas é fácil?
 - vejamos o diagrama de blocos de uma primitiva simples, o algoritmo de cifra simétrica DES....

Não parece ser muito fácil...



Fornecedores de segurança

- ❖ Em Java as operações criptográficas (ex: assinaturas digitais, sínteses de mensagens) são oferecidas através de um conjunto de **classes abstratas**
- ❖ A JVM fornece um arcabouço para concretização destas primitivas chamado JCA (*Java Cryptography Architecture*)
- ❖ Este arcabouço define um conjunto de classes e interfaces que podem ser usadas e concretizadas por diferentes fornecedores de segurança
 - A Sun fornece um com sua JVM, que concretiza as principais primitivas criptográficas
 - No entanto, qualquer um pode concretizar um fornecedor...
- ❖ Elementos fundamentais do JCA:
 - Fornecedor (provider) : um *package* ou conjunto de *packages* com concretizações de um conjunto de algoritmos criptográficos
 - Motor (engine): uma operação criptográfica abstracta
 - Ex: assinatura digital; síntese de mensagem
 - Algoritmo (algorithm): define como uma operação é executada; é uma concretização de um motor
 - Ex: MD5 ou SHA; RSA ou DSA
- ❖ Os **fornecedores** são a “cola” que associa **algoritmos** a **motores**

“Java Security”, cap. 8

Escolha de um fornecedor

- ❖ A classe `Security` percorre por ordem os fornecedores indicados no ficheiro de configuração `$JREHOME/lib/security/java.security`

- ❖ Exemplo:

- `security.provider.1=sun.security.provider.Sun`
- `security.provider.2=sun.security.rsa.SunRsaSign`
- `security.provider.3=com.sun.net.ssl.internal.ssl.Provider`
- `security.provider.4=com.sun.crypto.provider.SunJCE`
- `security.provider.5=sun.security.jgss.SunProvider`
- `security.provider.6=com.sun.security.sasl.Provider`
- `security.provider.7=org.jcp.xml.dsig.internal.dom.XMLDSigRI`
- `security.provider.8=sun.security.smartcardio.SunPCSC`
- `security.provider.9=sun.security.mscapi.SunMSCAPI`

Na versão 9 do java está em:
`$JREHOME/conf/security/java.security`

Ordem de
Procura

- ❖ Estas classes só podem ser carregadas pelo carregador de classes do sistema, logo têm de estar localizadas em `$JREHOME/lib/ext`

Exemplo

```
//gerar uma chave aleatória para utilizar com o AES
KeyGenerator kg = KeyGenerator.getInstance("AES");
kg.init(128);
SecretKey key = kg.generateKey();
```

```
Cipher c = Cipher.getInstance("AES");
c.init(Cipher.ENCRYPT_MODE, key);
```

```
// faltam buffered streams
```

```
FileInputStream fis;
FileOutputStream fos;
```

```
CipherOutputStream cos;
```

```
fis = new FileInputStream("c:\\tmp\\a.txt");
fos = new FileOutputStream("c:\\tmp\\a.cif");
```

```
cos = new CipherOutputStream(fos, c);
```

```
byte[] b = new byte[16];
while ((i=fis.read(b)) != -1) {
    cos.write(b, 0, i);
}
```

```
cos.close();
```

```
// byte[] keyEncoded = key.getEncoded();
```

```
//SecretKeySpec keySpec2 = new SecretKeySpec(keyEncoded, "AES");
```

```
//SecretKeySpec é subclasse de secretKey
```

```
//c.init(Cipher.DECRYPT_MODE, keySpec2);
```

❖ Geração de chaves

➤ KeyGenerator

❖ Chaves

➤ Key

➤ SecretKey

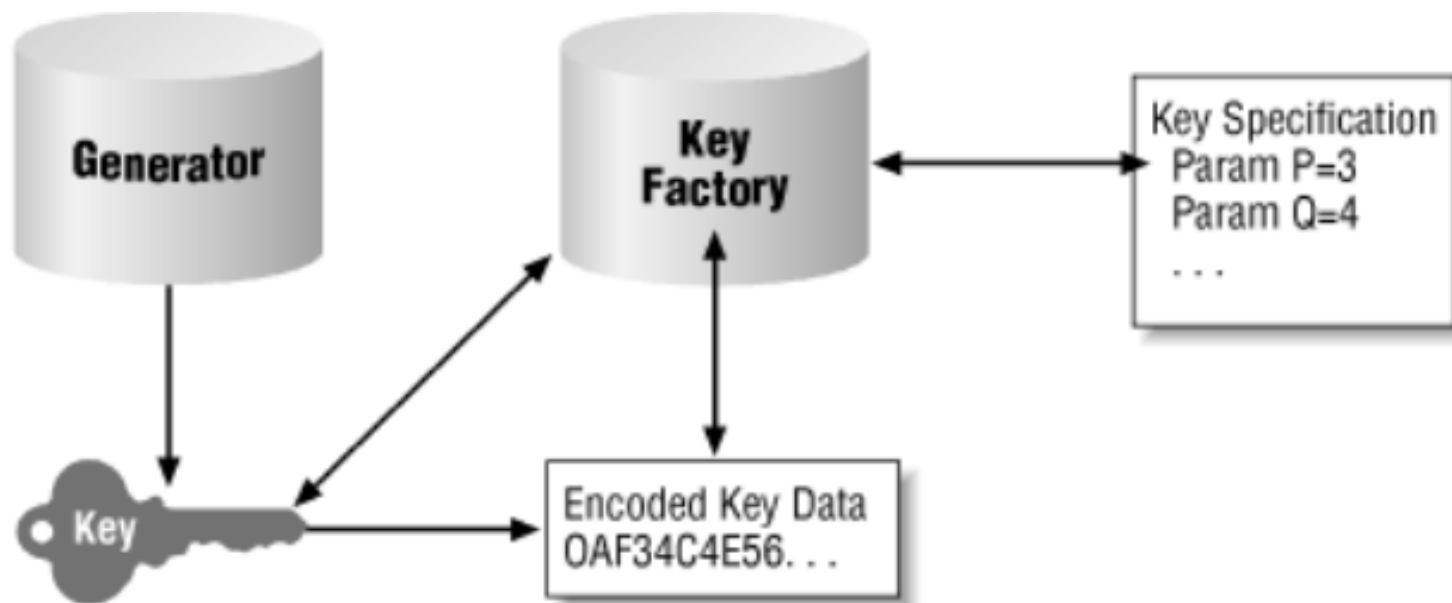
❖ Cifras

➤ Cipher

➤ Streams

Geração de Chaves

- ❖ Uma classe motor do tipo **Generator** cria as chaves sem quaisquer dados do utilizador
 - classe **KeyGenerator** cria chaves secretas
 - classe **KeyPairGenerator** cria pares de chaves público-privadas
- ❖ A classe motor **KeyFactory** traduz os objectos com as chaves numa representação externa (e.g., array de bytes ou uma especificação de chave)



Interface da classe *Key*

- ❖ A interface *java.security.Key* abstrai o conceito de chave
 - *public interface Key extends Serializable*
 - define uma única chave
 - precisa de ser *serializable* porque as chaves necessitam de ser transferidas entre diversas entidades
- ❖ *public String getAlgorithm()*
 - retorna uma *string* que indica que algoritmo gerou esta chave
- ❖ *public String getFormat()*
 - retorna uma *string* que descreve o formato usado para codificar a chave quando esta for transferida
- ❖ *public byte[] getEncoded()*
 - devolve um conjunto de bytes correspondentes à codificação da chave

(NOTA: reparar que **não existe** nada em relação à **descodificação da chave**)

Chaves secretas

- ❖ Existe uma interface no JCA que permite definir chaves secretas (*javax.crypto.SecretKey*)
 - *public interface **SecretKey** extends Key*
- ❖ As chaves secretas não têm qualquer tipo específico de informação de identificação, o que significa que esta interface é usada para facilitar a identificação dos tipos de objectos
- ❖ Embora sejam suportados diversos algoritmos simétricos, estes não definem interfaces próprias para as suas chaves (ao contrário do que acontece com a criptografia assimétrica)

Geração de Chaves Secretas

- ❖ A classe *javax.crypto.KeyGenerator* é usada para gerar estas chaves
 - *public class KeyGenerator*
 - gera chaves secretas para uma algoritmo de cifra simétrico

- ❖ Para se obter uma instância de um dado algoritmo usa-se:
 - *public static final KeyGenerator getInstance(String algorithm)*
 - *public static final KeyGenerator getInstance(String algorithm, String provider)*
 - O algoritmo pode ser algo como “DES” ou “AES” ou “HmacSHA1”

- ❖ Tendo-se a instância pode-se então chamar
 - *public final void init(SecureRandom sr)*
 - *public final void init(int strength)*
 - *public final void init(int strength, SecureRandom sr)*
 - inicializar o algoritmo (e.g., dimensão da chave e gerador de números aleatórios)
 - *public final SecretKey generateKey()*
 - cria nova chave (se chamado várias vezes gera diferentes chaves)

Exemplo

```
//gerar uma chave aleatória para utilizar com o AES
KeyGenerator kg = KeyGenerator.getInstance("AES");
kg.init(128);
SecretKey key = kg.generateKey();
```

```
Cipher c = Cipher.getInstance("AES");
c.init(Cipher.ENCRYPT_MODE, key);
```

```
// faltam buffered streams
```

```
FileInputStream fis;
```

```
FileOutputStream fos;
```

```
CipherOutputStream cos;
```

```
fis = new FileInputStream("c:\\tmp\\a.txt");
```

```
fos = new FileOutputStream("c:\\tmp\\a.cif");
```

```
cos = new CipherOutputStream(fos, c);
```

```
byte[] b = new byte[16];
```

```
while ((i=fis.read(b)) != -1) {
```

```
    cos.write(b, 0, i);
```

```
}
```

```
cos.close();
```

```
// byte[] keyEncoded = key.getEncoded();
```

```
//SecretKeySpec keySpec2 = new SecretKeySpec(keyEncoded, "AES");
```

```
//SecretKeySpec é subclasse de secretKey
```

```
//c.init(Cipher.DECRYPT_MODE, keySpec2);
```

❖ Geração de chaves

➤ KeyGenerator

❖ Chaves

➤ Key

➤ SecretKey

❖ Cifras

➤ Cipher

➤ Streams

Fábricas de Chaves (*KeyFactories*)

- ❖ São utilizadas para **converter** uma *Key* (formato opaco) em uma *KeySpec* (formato previamente especificado) e **vice-versa**
- ❖ Úteis para se **importar** ou **exportar** chaves entre diferentes formatos
- ❖ Existem no entanto algumas chaves que **apenas** podem ser produzidas com o auxílio das fábricas de chaves, por exemplo, uma **chave baseada numa *password***
- ❖ Existem duas representações externas que podem ser usadas para transmitir uma chave (ver especificação de chaves)
 - em um formato específico de codificação como **array de bytes** (e.g., base 64)
 - com os **parâmetros** usados para gerar a chave (e.g., uma *password*)

Utilização das *SecretKeyFactory*

- ❖ As classes *javax.crypto.SecretKeyFactory* são usadas nas conversões de chaves secretas, sendo o seu funcionamento exactamente igual ao das *KeyFactories*
- ❖ A classe é definida como
 - *public class SecretKeyFactory*
- ❖ Usam-se os seguintes métodos para se obter uma instância que permite exportar ou importar chaves geradas com um algoritmo
 - *public static final SecretKeyFactory getInstance(String alg)*
 - *public static final KeyFactory getInstance(String alg, String provider)*
- ❖ Métodos
 - *public final String getAlgorithm()*
 - devolve o nome do algoritmo suportado por esta fábrica

Utilização das *SecretKeyFactory*

❖ Métodos

- *public final SecretKey generateSecret(KeySpec ks)*
 - importa uma chave que se encontra especificada em *ks*
- *public final KeySpec getKeySpec(Key key, Class keySpec)*
 - exporta uma chave através da criação de uma especificação a partir da chave
- *public final Key translateKey(Key key)*
 - transforma uma chave entre um dado formato no formato desta fábrica (e.g., converter uma chave DSA do fornecedor XYZ para o da Sun)

❖ Existem várias classes *KeySpec*, entre elas *SecretKeySpec*, com métodos

- *SecretKeySpec(byte[] key, String Algorithm)*
 - para importar uma chave
- *byte[] getEncoded*
 - exporta dados

Exemplo

```
//gerar uma chave aleatória para utilizar com o AES
KeyGenerator kg = KeyGenerator.getInstance("AES");
kg.init(128);
SecretKey key = kg.generateKey();
```

```
Cipher c = Cipher.getInstance("AES");
c.init(Cipher.ENCRYPT_MODE, key);
```

```
// faltam buffered streams
```

```
FileInputStream fis;
FileOutputStream fos;
CipherOutputStream cos;
```

```
fis = new FileInputStream("c:\\tmp\\a.txt");
fos = new FileOutputStream("c:\\tmp\\a.cif");
```

```
cos = new CipherOutputStream(fos, c);
```

```
byte[] b = new byte[16];
while ((i=fis.read(b)) != -1) {
    cos.write(b, 0, i);
}
```

```
cos.close();
```

❖ Geração de chaves

➤ KeyGenerator

❖ Chaves

➤ Key

➤ SecretKey

❖ Cifras

➤ Cipher

➤ Streams

```
// byte[] keyEncoded = key.getEncoded();
```

```
//SecretKeySpec keySpec2 = new SecretKeySpec(keyEncoded, "AES");
```

```
//SecretKeySpec é subclasse de secretKey
```

```
//c.init(Cipher.DECRYPT_MODE, keySpec2);
```

Cifra

- ❖ Permite a cifra e decifra de dados que depois podem ser transmitidos numa mensagem ou armazenados num ficheiro
- ❖ Estas operações são realizadas por um conjunto de classes que se encontram disponíveis no fornecedor da Sun

“Java Security”, cap. 13

Classe *Cipher* (1)

- ❖ A classe *javax.crypto.Cipher* é um motor que fornece uma interface que possibilita a cifra e decifra de dados
 - *public class Cipher implements Cloneable*
- ❖ Como em todos os casos estudados anteriormente, esta classe precisa de ser instanciada antes de ser usada
- ❖ A indicação do nome da implementação é formada por

<nome do algoritmo>/<nome do modo de cifra>/<nome do tipo de padding>

- algoritmos: *AES, DES, DESede, Blowfish, RC4*, etc.
- modos: *ECB, CBC, CFB, OFB*
- *padding*: *PKCS5Padding, SSL3Padding, NoPadding*

Modo de cifra e tipo de *padding* são opcionais! Se não for indicado usa valor default!
NoPadding obriga a que os dados sejam múltiplos do tamanho do bloco de cifra!

Classe *Cipher* (2)

- ❖ Para se obter uma instância deve-se chamar com o nome do algoritmo (ex., AES, DES, DES/ECB/PKCS5Padding, etc.)
 - *public static Cipher getInstance(String algorithmName)*
 - *public static Cipher getInstance(String algorithmName, String provider)*
- ❖ Podem-se depois então invocar os seguintes métodos
 - *public final void init(int op, Key k)*
 - *public final void init(int op, Certificate c)*
 - *public final void init(int op, Key k, AlgorithmParameterSpec aps)*
 - *public final void init(int op, Key k, SecureRandom sr)*
 - *public final void init(int op, Certificate c, SecureRandom sr)*
 - *public final void init(int op, Key k, AlgorithmParameters ap)*
 - inicializa a cifra com informações relevantes (tipicamente uma chave) e
 - *op* define se a cifra será usada para cifrar (Cipher.ENCRYPT_MODE) ou decifrar (Cipher.DECRYPT_MODE) e
 - certos modos de cifra requerem um IV através *algorithm spec/param*
 - *public final Provider getProvider()*
 - retorna o fornecedor

Notar que também vão usar esta classe quando cifrarem com **criptografia assimétrica**

Classe *Cipher* (3)

- *public final byte[] update(byte[] input)*
- *public final byte[] update(byte[] input, int offset, int length)*
- *public final int update(byte[] input, int offset, int length, byte[] output)*
- *public final int update(byte[] input, int offset, int length, byte[] output, outoffset)*
 - cifrar ou decifrar dados, em que o resultado é devolvido num novo *array* ou no *output*; se os dados não forem múltiplos de bloco de cifra, o que sobra é armazenado internamente
 - estas funções podem ser chamadas mais do que uma vez
- *public final byte[] doFinal()*
- *public final int doFinal(byte[] output, int offset)*
- *public final byte[] doFinal(byte[] input)*
- *public final byte[] doFinal(byte[] input, int offset, int length)*
- *public final int doFinal(byte[] input, int offset, int length, byte[] output)*
 - semelhante ao anterior, mas só pode ser chamado uma vez (no final de vários *updates* ou sozinho)
- *public final int getOutputSize(int inputLength)*
 - devolve o tamanho total dos dados cifrados (incluindo padding)
- *public final byte[] getIV()*
 - retorna o IV que foi usado para inicializar a cifra

Exemplo

```
//gerar uma chave aleatória para utilizar com o AES
KeyGenerator kg = KeyGenerator.getInstance("AES");
kg.init(128);
SecretKey key = kg.generateKey();
```

```
Cipher c = Cipher.getInstance("AES");
c.init(Cipher.ENCRYPT_MODE, key);
```

```
// faltam buffered streams
```

```
FileInputStream fis;
```

```
FileOutputStream fos;
```

```
CipherOutputStream cos;
```

```
fis = new FileInputStream("c:\\tmp\\a.txt");
```

```
fos = new FileOutputStream("c:\\tmp\\a.cif");
```

```
cos = new CipherOutputStream(fos, c);
```

```
byte[] b = new byte[16];
```

```
while ((i=fis.read(b)) != -1) {
```

```
    cos.write(b, 0, i);
```

```
}
```

```
cos.close();
```

```
// byte[] keyEncoded = key.getEncoded();
```

```
//SecretKeySpec keySpec2 = new SecretKeySpec(keyEncoded, "AES");
```

```
//SecretKeySpec é subclasse de secretKey
```

```
//c.init(Cipher.DECRYPT_MODE, keySpec2);
```

❖ Geração de chaves

➤ KeyGenerator

❖ Chaves

➤ Key

➤ SecretKey

❖ Cifras

➤ **Cipher**

➤ Streams

Exemplo de cifra e decifra

Cifra uma string usando uma chave gerada e a seguir a decifra.

```
public class CipherTest {  
    public static void main(String args[]) throws Exception {  
        KeyGenerator kg = KeyGenerator.getInstance("DES");  
        Key key = kg.generateKey( );  
        Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");  
        c.init(Cipher.ENCRYPT_MODE, key);  
        byte input[] = args[0].getBytes( );  
        byte encrypted[] = c.doFinal(input);  
        byte iv[] = c.getIV( );    // obter o initialization vector por causa do CBC  
        IvParameterSpec ivs = new IvParameterSpec(iv);  
        c.init(Cipher.DECRYPT_MODE, key, ivs);  
        byte output[] = c.doFinal(encrypted);  
        System.out.println("The string was "+ new String(output));  
    }  
}
```

Streams de cifra

- ❖ Associa a um objecto de cifra uma *stream* de *input* ou *output*
 - à medida que os dados são escritos na *stream* são automaticamente cifrados
 - à medida que os dados são lidos da *stream* são automaticamente decifrados
- Note-se que isto permite a criação de canais mais seguros (ou pelo menos a confidencialidade pode ser satisfeita facilmente)!

Classes *CipherOutputStream* e *CipherInputStream*

- ❖ A classe *javax.crypto.CipherOutputStream* cifra os dados à medida que são escritos na stream
 - *public class CipherOutputStream extends FilterOutputStream*
- ❖ Os principais métodos desta classe são
 - *public CipherOutputStream(OutputStream outputStream, Cipher cipher)*
 - construtor que associa a um output stream um objecto de cifra (o objecto de cifra já deve estar inicializado)

... depois podem-se chamar os métodos habituais das streams (ex., *write*) ...
- ❖ A classe *javax.crypto.CipherInputStream* decifra dados à medida que são lidos de uma input stream
 - *public class CipherInputStream extends FilterInputStream*
- ❖ Os principais métodos desta classe são
 - *public CipherInputStream(InputStream is, Cipher c)*
 - construtor que associa a uma input stream um objecto de cifra

... depois usam-se os métodos habituais (ex., *read*) ...

Exemplo

```
//gerar uma chave aleatória para utilizar com o AES
KeyGenerator kg = KeyGenerator.getInstance("AES");
kg.init(128);
SecretKey key = kg.generateKey();
```

```
Cipher c = Cipher.getInstance("AES");
c.init(Cipher.ENCRYPT_MODE, key);
```

```
// faltam buffered streams
FileInputStream fis;
FileOutputStream fos;
CipherOutputStream cos;

fis = new FileInputStream("c:\\tmp\\a.txt");
fos = new FileOutputStream("c:\\tmp\\a.cif");

cos = new CipherOutputStream(fos, c);
byte[] b = new byte[16];
while ((i=fis.read(b) ) != -1) {
    cos.write(b, 0, i);
}
cos.close();
```

```
// byte[] keyEncoded = key.getEncoded();
```

```
//SecretKeySpec keySpec2 = new SecretKeySpec(keyEncoded, "AES");
//SecretKeySpec é subclasse de secretKey
//c.init(Cipher.DECRYPT_MODE, keySpec2);
```

❖ Geração de chaves

➤ KeyGenerator

❖ Chaves

➤ Key

➤ SecretKey

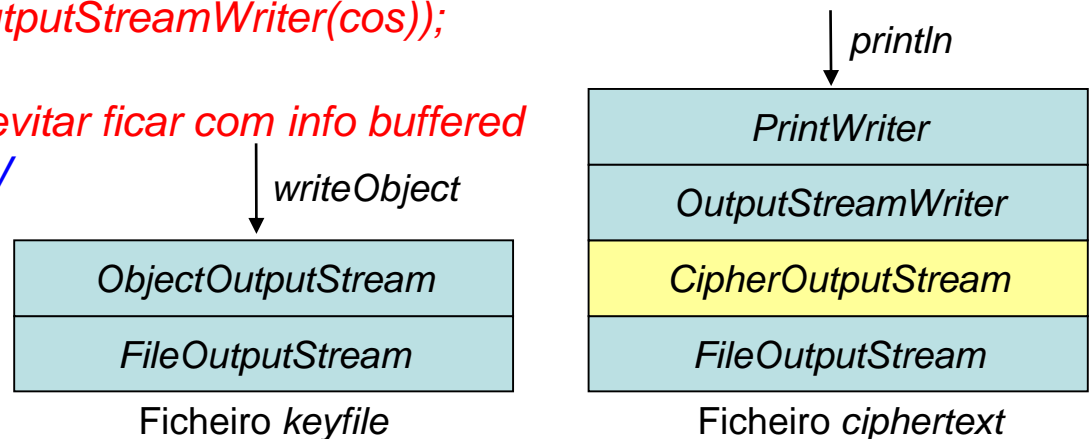
❖ Cifras

➤ Cipher

➤ **Streams**

Exemplo de Uso do *CipherOutputStream*

```
public class Send {  
    public static void main(String args[]) throws Exception {  
        KeyGenerator kg = KeyGenerator.getInstance("DES");  
        kg.init(new SecureRandom( ));  
        SecretKey key = kg.generateKey( );  
        SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");  
        Class spec = Class.forName("javax.crypto.spec.DESKeySpec");  
        DESKeySpec ks = (DESKeySpec) skf.getKeySpec(key, spec);  
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("keyfile"));  
        oos.writeObject(ks.getKey( )); // retorna byte[] com chave  
        Cipher c = Cipher.getInstance("DES/CFB8/NoPadding"); // CFB8 cifra um byte de cada vez  
        c.init(Cipher.ENCRYPT_MODE, key);  
        CipherOutputStream cos = new CipherOutputStream(new FileOutputStream("ciphertext"), c);  
        PrintWriter pw = new PrintWriter(new OutputStreamWriter(cos));  
        pw.println("Stand and unfold yourself");  
        pw.close( ); // fundamental fechar para evitar ficar com info buffered  
        oos.writeObject(c.getIV( )); // guardar IV  
        oos.close( ); // fundamental fechar  
    }  
}
```



Password-based Encryption

```
String password = "Come you spirits that tend on mortal thoughts";
byte[] salt = { (byte) 0xc9, (byte) 0x36, (byte) 0x78, (byte) 0x99, (byte) 0x52, (byte) 0x3e, (byte) 0xea, (byte) 0xf2 };

// Generate the key based on the password
PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray(), salt, 20); // pass, salt, iterations
SecretKeyFactory kf = SecretKeyFactory.getInstance("PBESWithHmacSHA256AndAES_128");
SecretKey key = kf.generateSecret(keySpec);

// ENCRYPTION: Lets check that the two keys are equivalent by encrypting a string
Cipher c = Cipher.getInstance("PBESWithHmacSHA256AndAES_128");
c.init(Cipher.ENCRYPT_MODE, key);
byte[] enc = c.doFinal("Ola Joana!");
byte[] params = c.getParameters().getEncoded(); // we need to get the various parameters (e.g. IV)

// DECRYPTION: Now lets see if we get the original string (NOTE: get key exactly as above)
AlgorithmParameters p = AlgorithmParameters.getInstance("PBESWithHmacSHA256AndAES_128");
p.init(params);
Cipher d = Cipher.getInstance("PBESWithHmacSHA256AndAES_128");
d.init(Cipher.DECRYPT_MODE, key, p);
byte [] dec = d.doFinal(enc);
```

Síntese (ou hash ou digest)

- ❖ As classes a seguir possibilitam a criação e verificação de sínteses seguras de mensagens
- ❖ Tanto o fornecedor por omissão da Sun como o JCE oferecem implementações destas classes

Classe *MessageDigest* (1)

- ❖ A classe *java.security.MessageDigest* oferece um conjunto de métodos que possibilitam a criação e verificação de uma síntese de mensagem
 - *public abstract class MessageDigest extends MessageDigestSpi*
- ❖ Como com todas as classes motor, utilizam-se os seguintes métodos para se obter uma instância da classe
 - *public static MessageDigest getInstance(String algorithm)*
 - *public static MessageDigest getInstance(String algorithm, String provider)*
- ❖ Tendo um referência para a instância podem-se chamar os métodos
 - *public void update(byte input)*
 - *public void update(byte[] input)*
 - *public void update(byte[] input, int offset, int length)*

para calcular uma síntese; chamadas consecutivas a estes métodos adicionam mais bytes aos dados que se quer calcular a síntese

Classe *MessageDigest* (2)

- *public byte[] digest()*
 - retorna a síntese dos dados acumulados até este momento (através do método *update*) e re-inicializa o estado interno do algoritmo para que uma nova síntese possa ser calculada
- *public byte[] digest(byte[] input)*
 - semelhante ao anterior, mas executa o *update(input)* e depois chama *digest()*
- *public int digest(byte[] output, int offset, int len)*
 - semelhante aos anteriores, mas a síntese é colocada em *output* no máximo *len* bytes (embora a maioria das implementações não devolvam sínteses parciais e por isso gera uma exceção caso não haja espaço suficiente)
- *public boolean isEqual(byte digestA[], byte digestB[])*
 - comparar se duas sínteses são iguais
- *public void reset()*
 - re-inicializar o estado interno, descartando a informação acumulada
- *public final int getDigestLength()*
 - retorna o número de bytes que tem a síntese devolvida por *digest()*
- *public final String getAlgorithm()*
 - devolve o nome do algoritmo (e.g., SHA)

Exemplo do uso de síntese

Abre um ficheiro *test*, e escreve uma mensagem “*This ... thee.*” juntamente com sua síntese criptográfica.

```
public class WriteFile {  
    public static void main(String args[]) throws Exception {  
        FileOutputStream fos = new FileOutputStream("test");  
        MessageDigest md = MessageDigest.getInstance("SHA");  
        ObjectOutputStream oos = new ObjectOutputStream(fos);  
        String data = "This have I thought good to deliver thee, "+  
            "that thou mightst not lose the dues of rejoicing " +  
            "by being ignorant of what greatness is promised thee.";  
        byte buf[] = data.getBytes( );  
        byte hash[] = md.digest(buf);  
        oos.writeObject(data);  
        oos.writeObject(hash);  
        fos.close();  
    }  
}
```

Exemplo de leitura e verificação de síntese

Abre o ficheiro *test* e verifica se não foi corrompido.

```
public class VerifyFile {  
    public static void main(String args[]) throws Exception {  
        FileInputStream fis = new FileInputStream("test");  
        ObjectInputStream ois = new ObjectInputStream(fis);  
        Object o = ois.readObject( );  
        if (!(o instanceof String)) {  
            System.out.println("error");  
            System.exit(-1);  
        }  
        String data = (String) o;  
        byte origDig[] = (byte []) ois.readObject( );    // devia-se validar "origDig" como em "data"  
        MessageDigest md = MessageDigest.getInstance("SHA");  
        if (MessageDigest.isEqual(md.digest(data.getBytes()), origDig))  
            //método isEqual faz apenas uma comparação byte a byte  
            System.out.println("valid");  
        else  
            System.out.println("Message was corrupted");  
        fis.close();  
    }  
}
```

MAC – *Message Authentication Code*

- ❖ Criação de um *hash* seguro que utiliza uma chave secreta partilhada entre os dois interlocutores
- ❖ O JCE por omissão da JVM oferece algumas realizações

Classe *Mac*

- ❖ A classe *javax.crypto.Mac* do JCE pode ser realizada com algoritmos de síntese, tendo os seguintes nomes *HmacMD5*, *HmacSHA1*, *HmacSHA256*, ...
(<http://docs.oracle.com/javase/7/docs/technotes/guides/security/StandardNames.html#Mac>)
- ❖ A interface e modo de uso é relativamente semelhante ao *MessageDigest*, sendo necessário obter uma instância (método estático *getInstance(...)*) antes de se poderem invocar os seguintes métodos
 - *public void init(SecretKey sk)*
 - *public void init(SecretKey sk, AlgorithmParameterSpec aps)*
 - inicializa o algoritmo com uma dada chave e parâmetros
 - *public byte[] doFinal()*
 - *public byte[] doFinal(byte[] input)*
 - *public void doFinal(byte[] output, int offset)*
 - calcula e devolve o MAC (usada em vez do *digest()*)
 - os demais métodos são semelhantes (e.g., o *update()*)

Exemplo em que se guarda um MAC

Escreve o texto no ficheiro *test* e adiciona um MAC gerado a partir de uma chave secreta.

```
public class WriteFileWithMac {  
    public static void main(String args[]) throws Exception {  
        FileOutputStream fos = new FileOutputStream("test");  
        Mac mac = Mac.getInstance("HmacSHA1");  
        SecretKey key = ... //obtém a chave secreta de alguma forma  
        mac.init(key);  
        ObjectOutputStream oos = new ObjectOutputStream(fos);  
        String data = "This have I thought good to deliver thee, .....";  
        byte buf[] = data.getBytes( );  
        mac.update(buf);  
        oos.writeObject(data);  
        oos.writeObject(mac.doFinal( ));  
        fos.close();  
    }  
}
```

```
// args[0] = alias; args[1] = password  
// args[2] = keystore location  
KeyStore ks = KeyStore.getInstance(  
    KeyStore.getDefaultType());  
FileInputStream fis = new  
    FileInputStream(args[2]);  
ks.load(fis, null);  
SecretKey key = ks.getKey(args[0],  
    args[1].toCharArray());
```

A verificação pode ser feita de forma semelhante!