Java Cryptography Architecture (JCA)

Para exercícios 6 e 7 do Trab 1
e para o Trab. 2

(tem muitas erros parvos)

ISEL – Instituto Superior de Engenharia de Lisboa
Rua Conselheiro Emídio Navarro, 1 | 1959-007 Lisboa

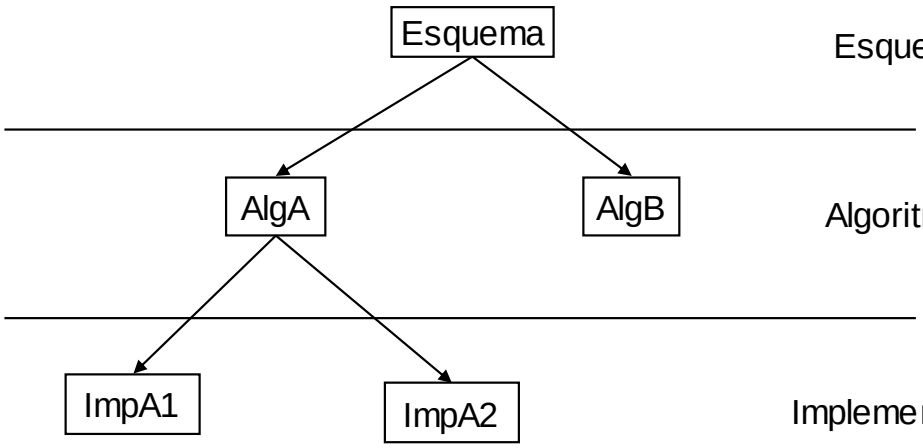
Desde os primórdios do Java, API não foi actualizada

Princípios de desenho

- Independência dos algoritmos e expansibilidade
  - Utilização de esquemas criptográficos, como a assinatura digital e a cifra simétrica, independentemente dos algoritmos que os implementam
  - Capacidade de acrescentar novos algoritmos para os mecanismos criptográficos considerados

- Independência da implementação e interoperabilidade
  - Várias implementações para o mesmo algoritmo
  - Interoperabilidade entre várias implementações
    - Por exemplo, assinar com uma implementação e verificar com outra
  - Acesso normalizado a características próprias dos algoritmos

Esquemas, algoritmos e implementações



formas pré-feitas

aqui vai dar dor de cabeça

Arquitectura

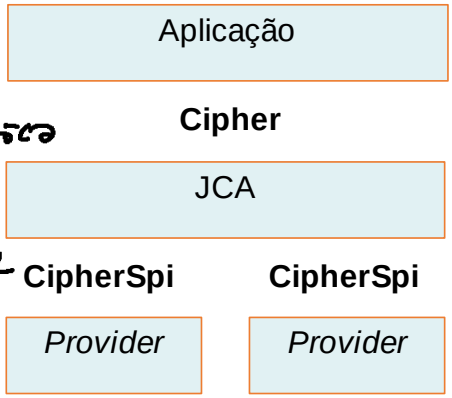
- Arquitectura baseada em:
  - CSP – *Cryptographic Service Provider*
    - *package* ou conjunto de *packages* que implementam um ou mais mecanismos criptográficos (serviços criptográficos)
  - *Engine Classes*
    - Definição abstracta (sem implementação) dum mecanismo criptográfico
    - A criação dos objectos é realizada através de métodos estáticos **getInstance**
  - *Specification Classes*
    - Representações normalizadas e transparentes de objectos criptográficos, tais como chaves e parâmetros de algoritmos.

2 formas de organizar os Providers

É preciso carregar os Provider primeiro no Maven / Gradle

Providers

- Fornecem a *implementação* para as *engines classes*
  - Implementam as classes abstractas <*EngineClass*>Spi, onde *EngineClass* é o nome duma *engine class*
- Classe **Provider** é base para todos os *providers*
- Instalação
  - Colocar *package* na *classpath* ou na directoria de extensões
  - Registar no ficheiro java.security
  - Em alternativa, usar a classe **Security**
- Classe **Security** → é melhor usar esta, é atomico ou noutro
  - Registo dinâmico de *providers*
  - Listagem de *providers* e algoritmos



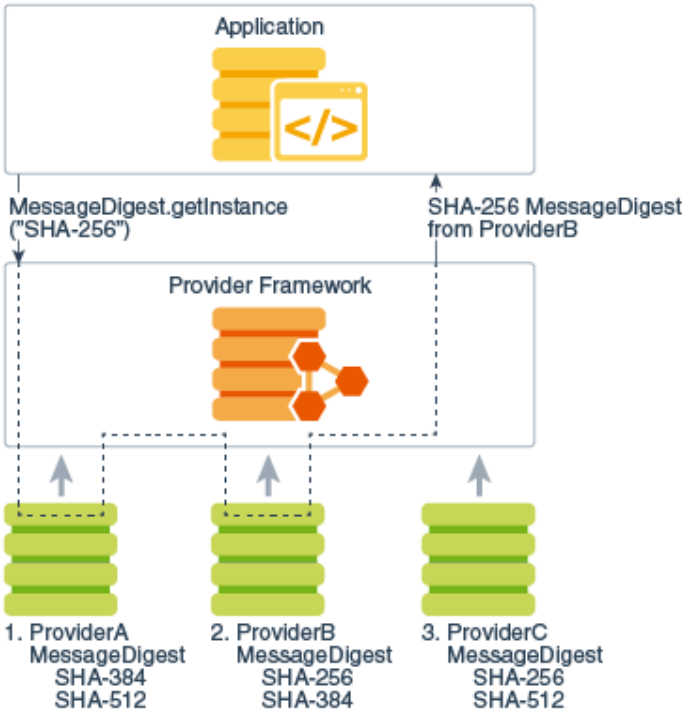
fazer testes
pois o código pode só explodir quando corrego isto

Classe do Hash é MessageDigest

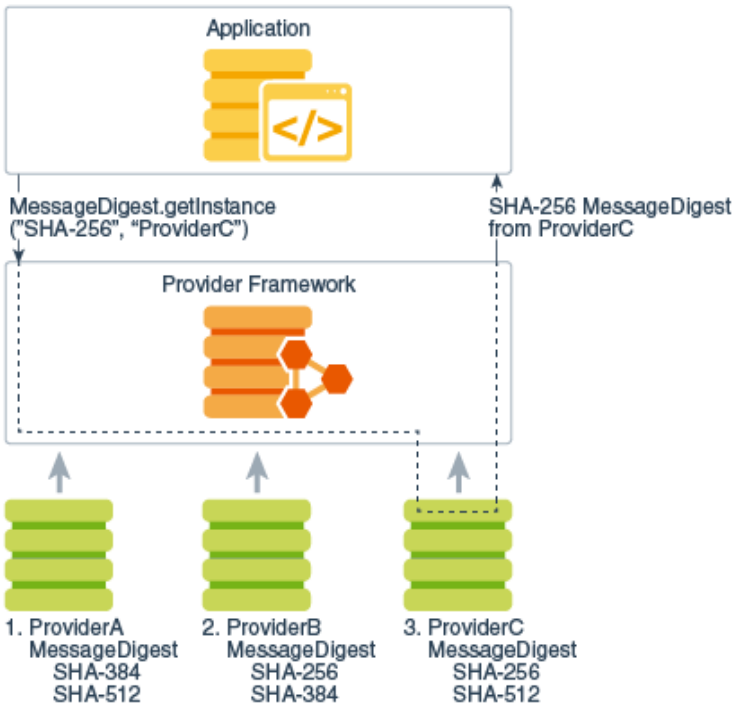
Para especificar o Provider

Providers – Exemplo com *Hash*

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




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



Fonte: https://docs.oracle.com/en/java/javase/11/security/java-cryptography-architecture-jca-reference-guide.html

User Providers da fornecida do SUN Microsystems

BouncyCastle - provider que tem muito coisa que os outros não têm
- tem backdoor para o governo australiano

Math.Random é previsível
SecureRandom é menos previsível
obrigatório usar no trabalho

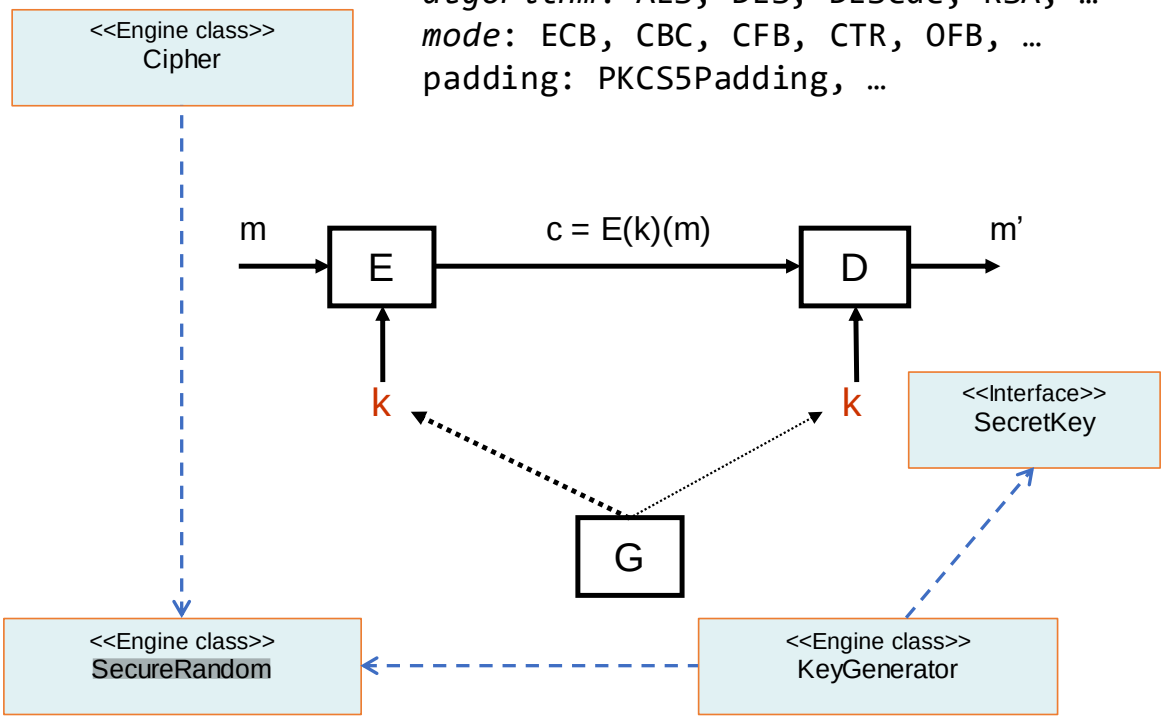
Engines classes – vão dar implementações

- Classes: **Cipher** (esquemas simétricos e assimétricos), **Mac**, **Signature**, **MessageDigest**, **KeyGenerator**, **KeyPairGenerator**, **SecureRandom**, …
- Métodos *factory* → Hash → Simétrica → Assimétrica → 1 pouco melhor que o simples Random
  - **static Cipher getInstance(String transformation)**
  - **static Cipher getInstance(String transformation, String provider)**
  - **static Cipher getInstance(String transformation, Provider provider)**
- Os algoritmos concretos e as implementações concretas (*providers*) são identificados por *strings*
- *Delayed Provider Selection* – pode dar problemas, bugs acontecem mais tarde no tempo
  - A Selecção do *provider* adequado é adiada até à iniciação com a chave
  - Permite a selecção do *provider* com base no tipo concreto da chave

7

Exemplo com cifra simétrica

Cipher
"algorithm/mode/padding" ou "algorithm"
algorithm: AES, DES, DESede, RSA, …
mode: ECB, CBC, CFB, CTR, OFB, …
padding: PKCS5Padding, …



Identificação do esquema
algorithm/mode/padding
ex: AES/ECB/PKCS5Padding

Exemplo do Código em Java

- Geração de chave simétrica aleatória a partir da SecureRandom

```
KeyGenerator keyGen = KeyGenerator.getInstance("AES");

// Opcional, se não passar um SecureRandom ao método init de keyGen
SecureRandom secRandom = new SecureRandom();

// Opcional
keyGen.init(secRandom);

SecretKey key = keyGen.generateKey(); // já tens key e IV

// Gera o objeto da cifra simetrica
Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");

// Associa a chave key a cifra
cipher.init(Cipher.ENCRYPT_MODE, key);

// Continuar com processo de cifra...
```

escolher Encriptar ou Desencriptar

Ver guia oficial do Oracle

Transformações normalizadas

- Ver apêndice A de "*Java Cryptography Architecture (JCA) Reference Guide*"
- **Cipher**
  - "algorithm/mode/padding" ou "algorithm"
  - *algorithm*: AES, DES, DESede, RSA, …
  - *mode*: ECB, CBC, CFB, CTR, OFB, …
  - *padding*: PKCS5Padding, PKCS1Padding, OAEPPadding
- **Mac**
  - hmac[MD5 | SHA1 | SHA256 | SHA384 | SHA512], …
- **Signature**
  - [MD5 | SHA1 | …]withRSA, SHA1withDSA, …
- **KeyGenerator**
  - AES, DES, DESede, Hmac[MD5 | SHA1 |…], …
- **KeyPairGenerator**
  - RSA, DSA, …

10

Classe Cipher

- Método **init** (várias sobrecargas)
  - Parâmetros: modo (cifra, decifra, *wrap* ou *unwrap*), chave, parâmetros específicos do algoritmo e gerador aleatório

- Métodos de cifra
  - **update**: **byte[]** → **byte[]** – continua a operação incremental
  - **doFinal: byte[]** → **byte[]** – finaliza a operação incremental
  - **wrap**: **Key** → **byte[]** – cifra chave
  - **unwrap: byte[], …** → **Key** – decifra chave

métodos principais

encapsulamento de chaves p/ envio

- Métodos auxiliares
  - **byte[] getIV()**
  - **AlgorithmParameters getParameters()**
  - …

doFinal é a forma correta de fazer Padding

-Para enviar chaves, fazer sempre o wrap

11

Cipher: Exemplo 1

- Cifrar uma mensagem texto simples

```
// Associa a chave key a cifra
cipher.init(Cipher.ENCRYPT_MODE, key);

// Mensagem a ser cifrada
String msg = new String("Mensagem secreta!");

// Mostra bytes da mensagem a ser cifrada
prettyPrint(msg.getBytes());

// Cifra mensagem com chave key
byte[] bytes = cipher.doFinal(msg.getBytes());

// Mostra os bytes em hexadecimal
prettyPrint(bytes);
```

Já traz key e IV

Modo de cifra.

- pode-se fazer logo para mensagens pequenas

Cipher: Exemplo 2

- Decifrar a mensagem cifrada do slide anterior

```
// Decifra com mesma chave da cifra
cipher.init(Cipher.DECRYPT_MODE, key);

byte[] bytes2 = cipher.doFinal(bytes);

// Mostra a mensagem original
System.out.println(new String(bytes2));
```

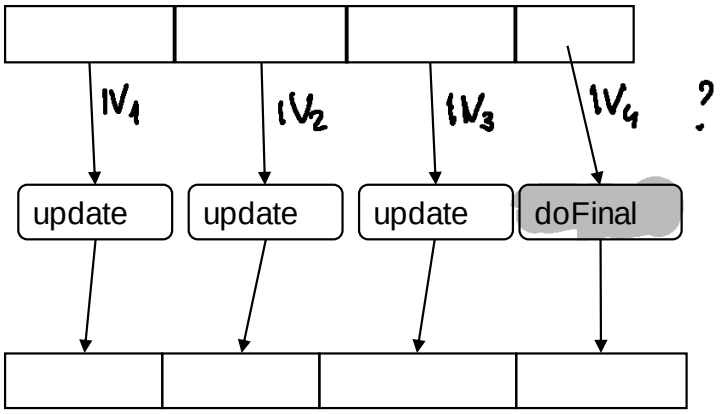
Usar a mesma chave do processo de cifra, que deve estar partilhada.

Modo de decifra.

```
// Imprime array de bytes em hexadecimal
static void prettyPrint(byte[] tag) {
    for (byte b: tag) {
        System.out.printf("%02x", b);
    }
    System.out.println();
}
```

Cipher: operação incremental

- Cifra de mensagens com grande dimensão ou disponibilizadas parcialmente
  - **Método update** recebe parte da mensagem e retorna parte do criptograma
  - **Método doFinal** recebe o final da mensagem e retorna o final do criptograma
- Nota: **E(k)(m1|| m2) ≠ E(k)(m1) || E(k)(m2)**



14

Cipher: Exemplo com update/doFinal

- Cifrar texto da entrada padrão parcialmente

```
// Obtém linha da entrada padrão in (Scanner) e adiciona
// quebra de linha que é removida pelo nextLine
nl = in.nextLine() + System.lineSeparator();

while (! System.lineSeparator().equals(nl)) {

    // Gera cifra parcial em tmp e concatena no criptograma c
    tmp = cipher.update(nl.getBytes());
    c = concatBytes(c, tmp);

    // Obtém próxima linha da entrada padrão
    nl = in.nextLine() + System.lineSeparator();
}

// Finaliza cifra e concatena bytes finais no criptograma c
tmp = cipher.doFinal();
c = concatBytes(c, tmp);

// Mostra bytes da mensagem cifrada
prettyPrint(c);
```

Nota: `concatBytes` é função (static) que deve ser implementada. Ela recebe dois arrays e retorna um novo array com a concatenação dos dois.

15

Streams (do Java)

- Classe **CipherInputStream : FilterInputStream**
  - Processa (cifra ou decifra) os *bytes* lidos através do *stream*
  - **ctor(InputStream, Cipher)**

- Classe **CipherOutputStream : FilterOutputStream**
  - Processa (cifra ou decifra) os *bytes* escritos para o *stream*

→ geração de hashes

- Class **DigestInputStream : FilterInputStream**
  - Processa (calcula o *hash*) os *bytes* lidos através do *stream*
  - **ctor(InputStream, MessageDigest)**
  - **MessageDigest getMessageDigest()**: método que retorna o *hash* associado ao *stream*

- Class **DigestOutputStream : FilterOutputStream**
  - Processa (calcula o *hash*) os *bytes* escritos para o *stream*

já controla os updates e doFinals

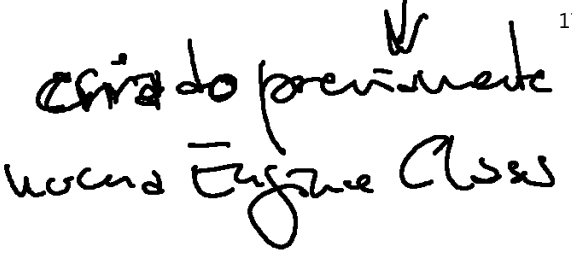
MD5 - Message Digest 5

CipherOutputStream: Exemplo

- Cifra texto da entrada padrão parcialmente e a apresenta na saída padrão

```
// Cifra e escreve o criptograma na saída padrão
CipherOutputStream cOutputStream = new CipherOutputStream(System.out, cipher);

// Obtém linha da entrada padrão in (Scanner) e adiciona quebra de linha
// que é removida pelo nextLine
nl = in.nextLine() + System.lineSeparator();

while (! System.lineSeparator().equals(nl)) {

    // Escreve a cifra da mensagem na saída padrão
    cOutputStream.write(nl.getBytes(), 0, nl.getBytes().length);

    // Obtém próxima linha da entrada padrão
    nl = in.nextLine() + System.lineSeparator();
}

cOutputStream.close();
```

Nota: A cifra finaliza na saída padrão no momento do `close` (ou *flush* da saída).



17

Parâmetros

- Parâmetros adicionais dos algoritmos
  - Exemplos: vector inicial (IV), dimensões das chaves

- Representação opaca: engine class **AlgorithmParameters**
- Representação transparente: classes que implementam a interface **AlgorithmParameterSpec**
  - Exemplos: **IvParameterSpec**, **RsaKeyGenParameterSpec**,…

- Geração: *engine class* **AlgorithmParameterGenerator**

- Transformação entre representações transparentes e representações opacas: métodos de **AlgorithmParameters**

Cada Engine Class tem conjuntos diferentes de parâmetros

2 vamos ter de ter de cabeça — IVs, Chaves RSA

É necessário restaurar os parâmetros e as chaves

18

Parâmetros: Exemplo (transparente)

- Gerar chave e IV a partir de valores estáticos

```
// Gera os bytes para o vetor de bytes correspondente a chave
byte[] keyBytes = {0x01, 0x23, 0x45, 0x67, (byte)0x89, (byte)0xab, (byte)0xcd,
(byte)0xef, 0x01, 0x23, 0x45, 0x67, (byte)0x89, (byte)0xab, (byte)0xcd, (byte)0xef};

// Gera os bytes para o vetor de bytes correspondente ao IV
byte[] ivBytes  = {(byte)0xef, (byte)0xcd, (byte)0xab, (byte)0x89, 0x67, 0x45, 0x23,
0x01, (byte)0xef, (byte)0xcd, (byte)0xab, (byte)0x89, 0x67, 0x45, 0x23, 0x01};

// Gera chave a partir do vetor de bytes (valor fixo, não aleatório)
SecretKey key = new SecretKeySpec(keyBytes, "AES");

// Gera IV a partir do vetor de bytes (valor fixo, não aleatório)
IvParameterSpec iv = new IvParameterSpec(ivBytes);

// Gera o objeto da cifra simetrica
Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

// Associa a chave key a cifra
cipher.init(Cipher.ENCRYPT_MODE, key, iv);

// Continuar com a cifra de uma mensagem...
```

Classes que representam os respetivos objetos de forma **transparente**.

ISEL

ter atenção ao tamanho do IV

19

Chaves

- Interface **Key**
  - **String getAlgorithm()**
  - **byte[] getEncoded()**
  - **String getFormat()**

- Interfaces **SecretKey**, **PublicKey** e **PrivateKey**
  - Derivam de **Key**
  - Não acrescentam métodos (*marker interfaces*)

- Classe concreta **KeyPair**
  - Contém uma **PublicKey** e uma **PrivateKey**

- Geração através das *engine classes* **KeyGenerator** e **KeyPairGenerator**

20

Par de Chaves: Exemplo de geração

- Representação opaca das chaves

```
// Cria objeto KeyPair
KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");

// Inicia o tamanho da chave
keyPairGen.initialize(2048);

// Gera o par de chaves
KeyPair pair = keyPairGen.generateKeyPair();

// Obtém a chave privada
PrivateKey privKey = pair.getPrivate();

// Obtém a chave pública
PublicKey publicKey = pair.getPublic();

System.out.println("A chave pública em hexadecimal:");
prettyPrint(publicKey.getEncoded());
```

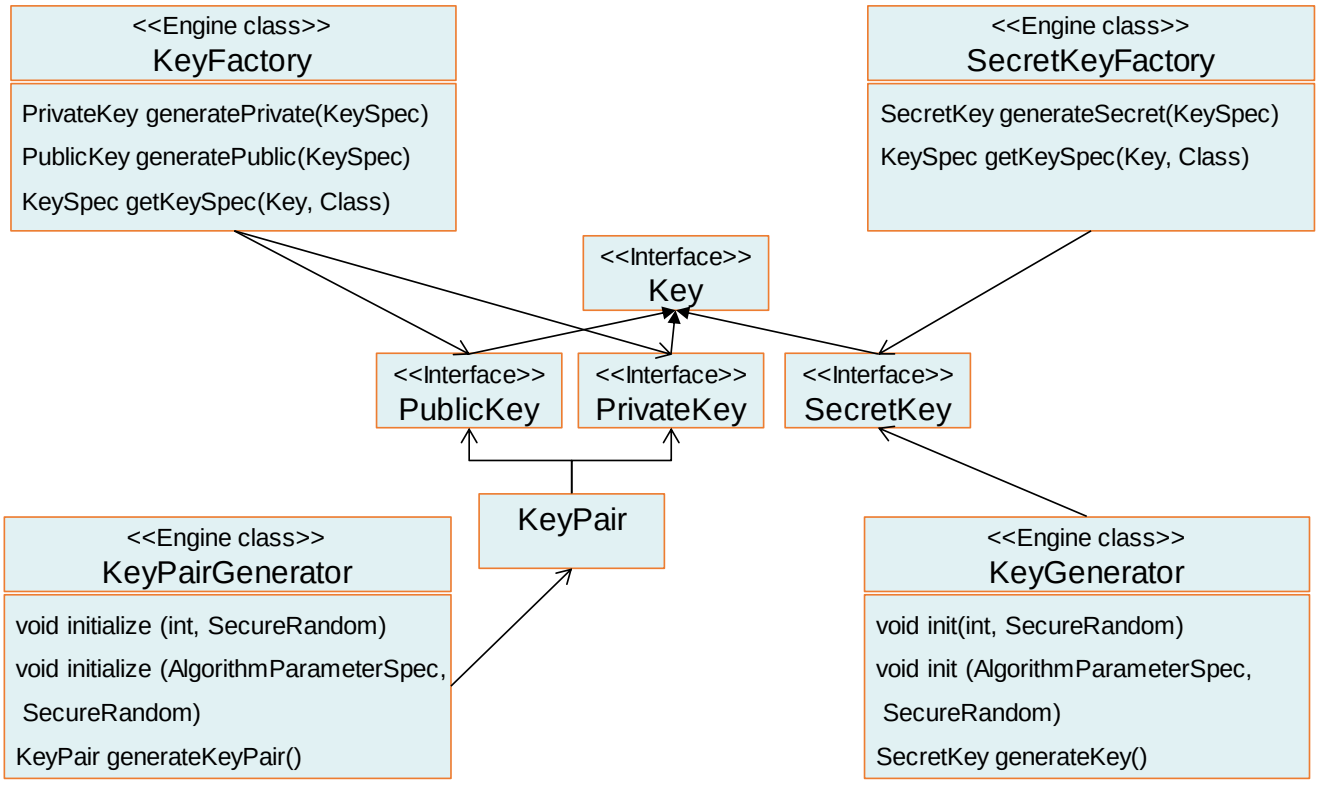
Chaves: Representações

Opacas e transparentes

- Chaves opacas: representações de chaves **sem** acesso aos seus constituintes
  - Derivadas da interface **Key** → tem chave e IV
  - Específicas de cada *provider*
  - Geradas pelas *engine classes* **KeyGenerator** e **KeyPairGenerator**
- Chaves transparentes: representações de chaves **com** acesso aos seus constituintes
  - Derivadas da interface **KeySpec** → é necessário configurar chave parâmetros e IV
  - Os packages **java.security.spec e javax.crypto.spec** definem um conjunto de classes ***<nome>*Spec** com interface normalizada para o acesso aos constituintes das chave de diversos algoritmos.
  - Exemplos: **RsaPublicKeySpec**, **DESKeySpec**, **SecretKeySpec**, …
- **KeyFactory** – *engine class* para conversão entre representações opacas e transparentes

Carlos - formatação correta do JWT

Chaves, geradores e fábricas



Chaves Transparentes: Exemplo

- Obter dados transparentes das chaves RSA
  - Supor par de chaves RSA já gerado

```
// KeyFactory para obter detalhes das chaves
KeyFactory kf = KeyFactory.getInstance(keyPairGen.getAlgorithm());

// Conversões do objeto opaco para o transparente
RSAPrivateKeySpec privKeySpec = kf.getKeySpec(privKey,
RSAPrivateKeySpec.class);
RSAPublicKeySpec publicKeySpec = kf.getKeySpec(publicKey,
RSAPublicKeySpec.class);

// Mostrar dados transparentes das chaves
System.out.println("Private Modulus: " + privKeySpec.getModulus());
System.out.println("Private Exponent: " +
privKeySpec.getPrivateExponent());
System.out.println("Public Modulus: " + publicKeySpec.getModulus());
System.out.println("Public Exponent: " +
publicKeySpec.getPublicExponent());
```

Tem de se passar a classe correta

Tudo isto depende dos Providers, mudo é certo.

Classe Mac

- Método **init** (várias sobrecargas)
  - Parâmetros: chave e parâmetros específicos do algoritmo

- Métodos de geração de marca
  - **update**: **byte[]** → **void** – continua a operação incremental
  - **doFinal: byte[]** → **byte[]** – finaliza a operação incremental, retornando a marca

- Métodos auxiliares
  - **int getMacLength()**
  - …

Classe Mac: Exemplo

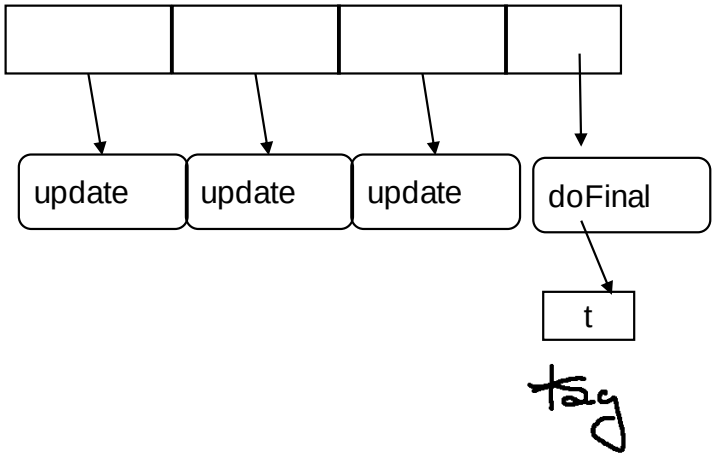
- Gerar HMAC-SHA256 de uma mensagem texto

```
byte[] someImportantMessage = "This is a very important message";

// Gerador de chaves HMAC-SHA256
KeyGenerator secretKeyGenerator = KeyGenerator.getInstance("HmacSHA256");

// Gera a chave simétrica
SecretKey key = secretKeyGenerator.generateKey();

// Obtém objeto MAC e inicia com a chave
Mac mac = Mac.getInstance("HmacSHA256");
mac.init(key);

// Computa o HMAC da mensagem
byte[] tag = mac.doFinal(someImportantMessage.getBytes());

prettyPrint(tag);
```


Calcular o MAC da mesma forma que se calcula a cifra, ... update, doFinal

Mac: operação incremental

- MAC de mensagens com grande dimensão ou disponibilizadas parcialmente
  - Método **update** recebe parte da mensagem
  - Método **doFinal** recebe o final da mensagem e retorna a marca

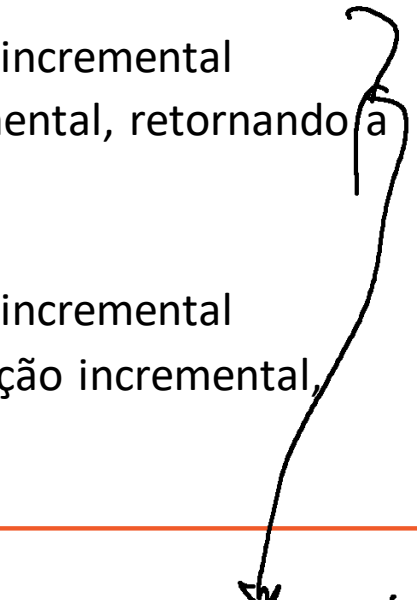


- Um pouco diferente das outras classes, não tem init e sim initSign e initVerify diferente
- Igual ao MAC

Classe Signature

- Método **initSign** (várias sobrecargas)
  - Parâmetros: chave privada e gerador aleatório
- Método **initVerify** (várias sobrecargas)
  - Parâmetros: chave pública

- Métodos de geração de assinatura
  - **update**: **byte[]** → **void** – continua a operação incremental
  - **sign: void** → **byte[]** – finaliza operação incremental, retornando a assinatura
- Métodos de verificação de assinatura
  - **update**: **byte[]** → **void** – continua a operação incremental
  - **verify**: **byte[]** → **{true,false}** – finaliza a operação incremental, retornando a validade da assinatura



Já não existe doFinal

Mas sim sign ou verify

Classe Signature: Assinatura (I)

- Assinar mensagem texto com SHA256 with RSA
  - Primeira parte: gerar chaves assimétricas

```
String m = "Mensagem texto a ser assinada";

// Uso de assinatura digital baseada no RSA
KeyPairGenerator kg = KeyPairGenerator.getInstance("RSA");

// Como exemplo, gerar uma chave RSA de 4096 bits
kg.initialize(4096);

// Gerar o par de chaves
KeyPair keyPair = kg.generateKeyPair();

// Obter as chaves pública e privada
PublicKey pubKey = keyPair.getPublic();
PrivateKey privKey = keyPair.getPrivate();

// Continua...
```

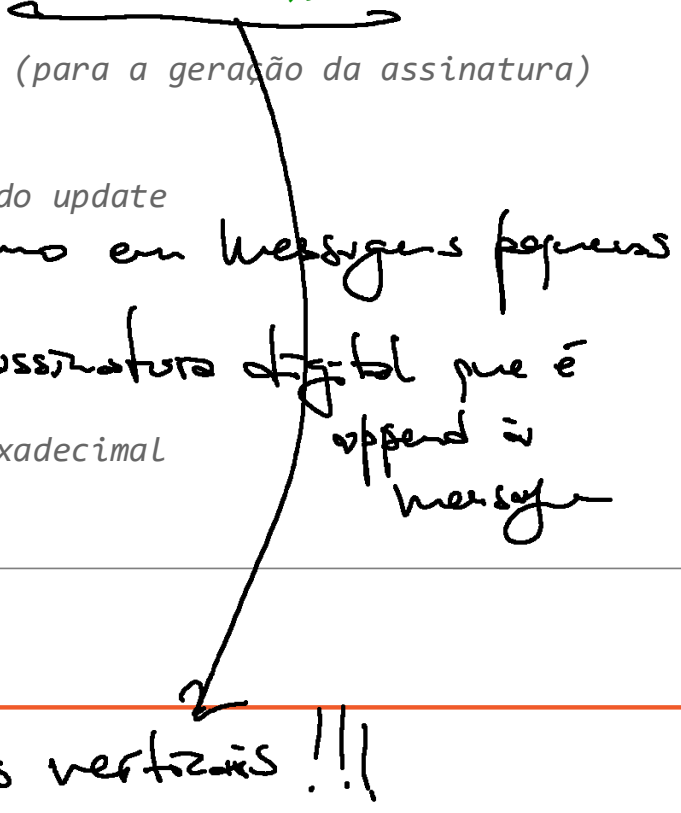

Uma assinatura tem 2 fases:

- criar hash
- assinar hash

Classe Signature: Assinatura (II)

- Segunda parte: processo de assinatura

```
// Obter uma instância do algoritmo de assinatura SHA256withRSA
Signature sign = Signature.getInstance("SHA256withRSA");

// Iniciar o objeto com a chave privada (para a geração da assinatura)
sign.initSign(privKey);

// Solicitação da assinatura com o método update
sign.update(m.getBytes());

// Finalizar a assinatura em si
byte[] s = sign.sign();

// Mostrar os bytes da assinatura em hexadecimal
System.out.printf("signature: ");
prettyPrint(s);
```



Já nos vai dar boas vertigens !!!

Classe Signature: Verificação

- Processo de verificação da assinatura
  - Supor chave pública partilhada
  - Verificar a mensagem *m* do processo do slide anterior

```
// Precisa-se inicializar o objeto em modo de verificação
// utilizando a porção pública da chave
sign.initVerify(pubKey);

// Associar mensagem a ser verificada ao objeto de verificação
sign.update(m.getBytes());

// Finalmente, usar o método verify() para verificar a assinatura
if (sign.verify(s)) System.out.println("Signature matches!");
else System.out.println("Signature does not match!");
```

Signature: operação incremental

- Assinatura/verificação de mensagens com grande dimensão ou disponibilizadas parcialmente
  - **update** recebe as parte da mensagem
  - **sign/verify** produz/verifica a assinatura