Cryptographie et Java

Caractéristiques idéales d'une communication sécurisée : un peu de vocabulaire…

Cryptologie (« science du secret ») = ensemble des techniques donnant à des données/informations les propriétés suivantes :

- **Confidentialité** = les données transmises doivent être incompréhensibles pour les personnes autres que l'émetteur (E) et le récepteur (R) visé → utilisation de méthodes de cryptographie (« écriture secrète »)
- **Authentification** = celui qui reçoit le message doit être certain de l'origine de celui-ci → utilisation de la signature électronique
- **Intégrité** = celui qui reçoit le message doit être certain que les données n'ont pas été modifiées de puis leur envoi par l'émetteur
- **Non-répudiation** = l'émetteur ne doit pas pouvoir nier avoir envoyé le message

Crypto-analyse = ensemble des techniques permettant de rendre clair le contenu d'un message crypté

Cryptage (= « chiffrement ») = opération consistant à transformer un **message clair** en **message crypté** (= qui n'est pas compréhensible pour les personnes autres que E et R)

Décryptage = opération consistant à retrouver le message clair contenu dans un message crypté

Bref historique et introduction aux notions de base

Code de César : principe de substitution

Principe : chaque lettre d'un message est remplacée par celle qui se trouve X caractères plus loin.

Exemple pour X = 3 :

caractère clair	a	b	c	d	e	f	…	x	y	z
caractère crypté	d	e	f	g	h	i	…	a	b	c

Cryptage du message « vilvens » :

message clair (plain text)	v	i	l	v	e	n	s
message crypté (cipher text)	y	l	o	y	h	q	v

Technique utilisée : **substitution** de caractères → **algorithme de cryptage**

Valeur de X → **clé de cryptage/décryptage**

L'algorithme est « public » (connu de tous), c'est la connaissance de la clé (qui doit rester secrète → on parle aussi de **clé secrète**) qui est fondamentale.

Inconvénients :

- Longueur message crypté = longueur message clair
- Une lettre est toujours remplacée par la même lettre → facilité de casser le cryptage
- Logique arithmétique simple → facilité de casser le cryptage

Variante : on peut imaginer en plus, avant substitution, que l'ordre des lettres soit modifié → **principe de transposition de caractères**.

Exemple : « permuter les lettres de 3 couples successifs, laisser la suivante inchangée, puis recommencer » :

message clair (plain text)	c	l	a	u	d	e	v	i	l	v	e	n	s
message crypté (cipher text)	l	c	u	a	e	d	v	l	i	e	v	s	n

Des clés secrètes plus élaborées

Principe : imaginer des mécanismes de substitutions plus compliquées utilisant des clés plus sophistiquées comme un mot

Exemple d'algorithme : utiliser un tableau de substitution qui associe les premières lettres de l'alphabet aux lettres de la clé (sans répétition). Si la clé est « white is the sky » :

clair	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
crypté	**w**	**h**	**i**	**t**	**e**	**s**	**k**	**y**	z	a	b	c	d	f	g	j	l	m	n	o	p	q	r	u	v	x

La clé doit être générée d'une certaine manière → on parle d'**algorithme de génération de clé**

Variantes : chiffrements polyalphabétiques (Alberti, 15ème siècle ; Vigenère, 16ème siècle)

→ on change la table de substitution en cours de cryptage

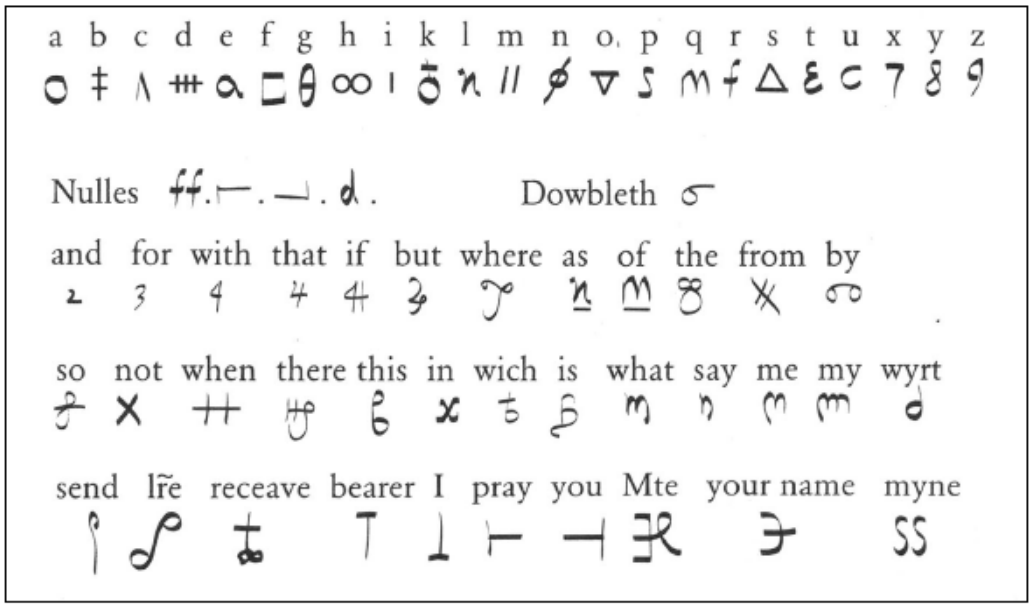
→ une lettre claire n'est pas toujours cryptée de la même manière à chaque occurrence

Chiffrement ou codage ?

On peut imaginer qu'un mot entier soit remplacé par un autre mot ou symbole

Exemple (Renaissance et époque classique) : Code de Mary Stuart :

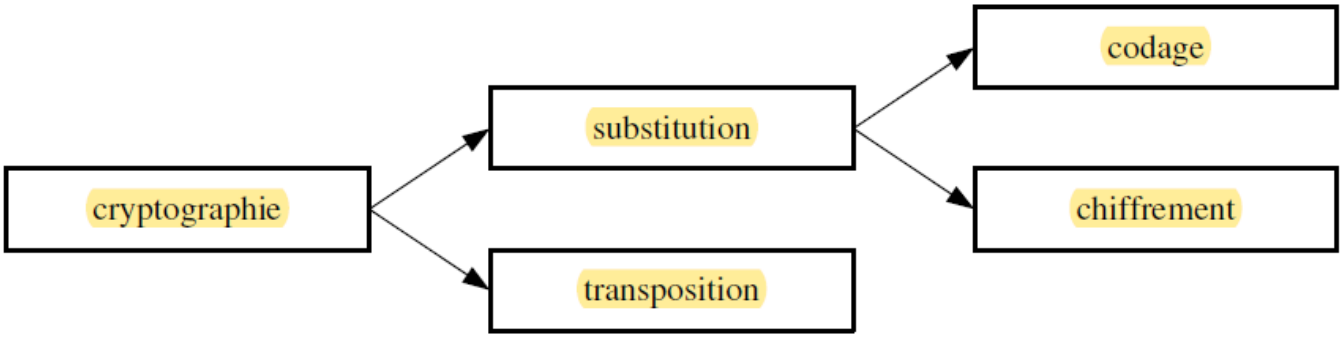
- Les lettres sont remplacées par des caractères spéciaux
- Certains mots sont remplacés par d'autres caractères spéciaux
- Certains caractères spéciaux ne servent à rien, sauf à embrouiller



On parle alors de

- **Codage [*code*]** = on remplace un mot par un autre mot
- **Chiffrement** = **Chiffre [*cipher*]** = on remplace une lettre par une lettre

Résumé :



Substitution + Transposition + découpage en blocs

Principe :

1. Découper le message clair en **blocs de longueur fixe**

2. Transposer les blocs selon une certaine règle utilisant une clé

Exemple : transposition simple à clé → clé = « latin » et message clair = « évacuer les bases ». On découper le message en blocs de 5 lettres et on les aligne en colonnes :

l	*a*	*t*	*i*	*n*
e	v	a	c	u
e	r	l	e	s
b	a	s	e	s

On ré-écrit les lettres de la clé par ordre alphabétique et on permute les colonnes :

a	*i*	*l*	*n*	*t*
v	c	e	u	a
r	e	e	s	l
a	e	b	s	s

Message crypté = « vceuareeslaebss »

Problème : Si le message clair est « vilvens sera absent », cela donne

l	*a*	*t*	*i*	*n*
v	i	l	v	e
n	s	s	e	r
a	a	b	s	e
n	t			

- ➔ Le dernier bloc est incomplet
- ➔ Il faut le compléter judicieusement (0 ? autre chose ?)
- ➔ Utilisation d'**algorithmes de remplissage [*padding*]**

On peut combiner à cela le **principe de substitution**.

Exemple : Codage ADFGX utilisés par les Allemands pendant la 1ère guerre mondiale

Chaque lettre du message clair est remplacée par deux lettres (symbole appelé « bigramme ») à l'aide du tableau suivant (appelé carré de Polybe) :

	A	D	F	G	X
A	c	q	h	b	l
D	o	e	i,j	s	g
F	d	n	k	v	y
G	t	f	x	r	z
X	m	w	u	p	a

Si on veut crypter « vilvens sera absent » avec comme clé « latin » :

msg clair (plain text)	v	i	l	v	e	n	s	s	e	r	a	a	b	s	e	n	t
msg crypté (ciph	FG	DF	AX	FG	DD	FD	DG	DG	DD	GG	XX	XX	AG	DG	DD	FD	GA

Le message crypté est alors rangé dans une grille et les colonnes sont permutées :

l	*a*	*t*	*i*	*n*
FG	DF	AX	FG	DD
FD	DG	DG	DD	GG
XX	XX	AG	DG	DD
FD	GA			

→

a	*i*	*l*	*n*	*t*
DF	FG	FG	DD	AX
DG	DD	FD	GG	DG
XX	DG	XX	DD	AG
GA		FD		

Le message crypté est finalement :

DF FG FG DD AX DG DD FD GG DG XX DG XX DD AG GA FD

Cryptage électromécanique

Enigma (2ème guerre mondiale) : machine électromécanique complexe

Cryptage « cassé » par **Alan Turing**, un des pionniers et fondateurs de l'informatique moderne

Bases des techniques cryptographiques

Clés et algorihthmes

Schéma général d'une transmission codée :

1. Génération d'un **message clair [plaintext]** par un émetteur (E)
2. **Cryptage** = **chiffrement [encryption]** : transformation du message clair en un **message crypté** (= codé = chiffré) → La méthode utilisée s'appelle un **chiffre [cipher]** et utilise un **algorithme de chiffrement à clé(s)**
3. **Transmission** du message crypté de l'émetteur (E) vers un récepteur (R) par le réseau (par exemple)
4. **Décryptage** = **déchiffrement** du message crypté par le récepteur (R) en utilisant l'algorithme de décryptage et la clé associée.

La notion de **clé** est fondamentale :

Une clé est un bloc de d'informations (des bits) permettant un chiffrement ou un déchiffrement

Une clé peut être produite à l'aide d'un **algorithme de génération de clé**.

On distingue deux types d'algorithmes de cryptage :

- **Algorithmes symétriques ou à clé secrète** : la même clé est utilisée au chiffrement et au déchiffrement → **confidentialité**

- **Algorithmes asymétriques ou à clé publique** : on utilise une paire de clés : une pour chiffrer (appelée **clé publique [public key]**) et une autre pour déchiffrer (appelée **clé privée [private key]**) → **confidentialité** + **authentification** → on parle de **PKI** (**P**ublic **K**ey **I**nfrastructure).

Modes de chiffrement

Il s'agit de la manière dont les blocs d'un texte clair correspondent à des blocs de texte chiffré

- **Chiffrement de bloc** : transforme un bloc de données claires d'une taille préalablement fixée en un bloc de données chiffrées de même taille
- **Chiffrement de flux** : opère sur les bits ou des unités de petites tailles et les transforme de manière variable au cours du temps

Mode de chiffrement de bloc

Il faut

1) **Découper** le message clair **en blocs** de taille acceptée par l'algorithme utilisé
2) **Chiffrer chaque bloc** → ceci se fait en utilisant un **mode de chaînage** dont les plus courant sont
    - **ECB** (**E**lectronic **C**ode **B**ook) : un bloc de texte clair se chiffre en un bloc de texte chiffré, indépendamment des autres blocs → 2 blocs identiques produisent les mêmes blocs chiffrés
    - **CBC** (**C**ipher **B**loc **C**haining) : chaque bloc de texte clair est combiné par XOR avec le bloc de texte chiffré précédent → un « vecteur d'initialisation » (appelée **IV**) fournit le bloc chiffré pour le premier bloc clair → 2 blocs identiques ont peu de chance de produire les mêmes blocs chiffrés

Si le message à coder a une taille non multiple de la taille d'un bloc, il faut le compléter avec des **caractères de remplissage [padding]** → les paddings les plus connus sont ceux des recommandations

- **PKCS#5** (algorithme symétrique **DES**)
- **PKCS#7** (algorithme asymétrique **RSA**)

Les **PKCS** (**P**ublic **K**ey **C**ryptography **S**tandards) sont un ensemble de spécifications publiées par RSA Data Security.

Les chiffrements symétriques

Principe général :

La même clé est utilisée pour chiffrer et déchiffrer un message

Le secret repose donc sur la connaissance de la clé → **algorithme à clé secrète**

Exemple :

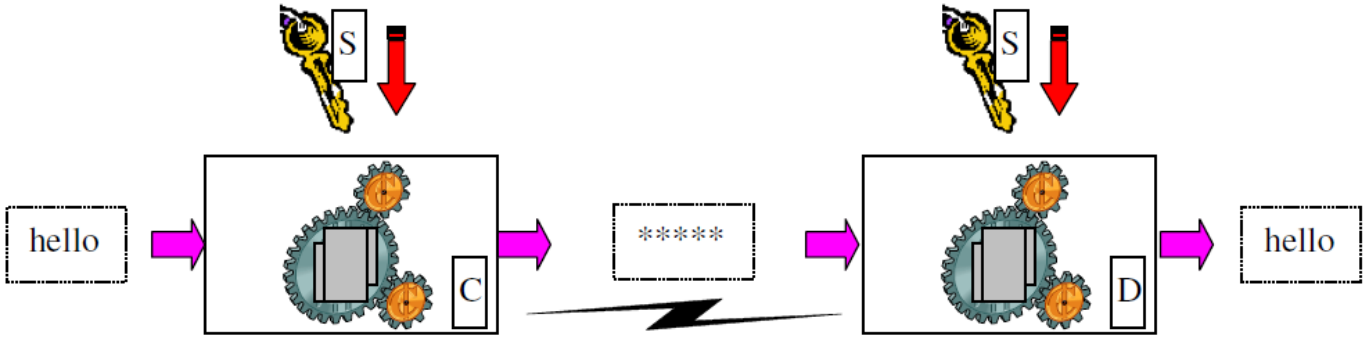
Message à coder : « hello » (codes ASCII : 104 – 101 – 108 – 108 – 111)

Clé secrète : « sidney » (codes ASCII : 115 – 105 – 100 – 110 – 101 – 121)

Algorithme : Application d'un XOR sur le code ASCII de chaque lettre du message avec celui d'une lettre de la clé

chiffrement :

message clair	"hello"	0110 1000	0110 0101	0110 1100	0110 1100	0110 1111
clé	"sidney"	0111 0011	0110 1001	0110 0100	0110 1110	0110 0101
XOR						
message crypté	*****	0001 1011	0000 1100	0000 1000	0000 0010	0000 1010



(S=clé Secrète; C=Chiffrement; D=Déchiffrement)

déchiffrement :

message crypté	*****	0001 1011	0000 1100	0000 1000	0000 0010	0000 1010
clé	"sidney"	0111 0011	0110 1001	0110 0100	0110 1110	0110 0101
XOR						
message obtenu	"hello"	0110 1000	0110 0101	0110 1100	0110 1100	0110 1111

Quelques algorithmes courants :

- **DES** (**D**ata **E**ncryption **S**tandard) : standard du gouvernement américain, blocs de 64 bits, clé secrète de 56 bits (en fait 64 bits mais 8 bits servent à un contrôle d'intégrité de la clé)
- **TripleDES** : triple application du DES utilisant une 2ème clé
- **Blowfish** : développé pour les machines 32 bits, blocs de 64 bits, clé de taille variable (<= 448 bits), plus rapide que DES
- **IDEA** (**I**nternational **D**ata **E**ncryption **A**lgorithm) : conçu pour résister aux techniques de crypto-analyse évoluées, blocs de 64 bits, clé de 128 bits
- **SAFER** (**S**ecure **A**nd **F**ast **E**ncryption **R**outine)
- **Rijndael** (concepteurs belges **Rij**nen et **Dae**nen) et **AES** (**A**dvanced **E**ncryption **S**tandard) : algorithme sélectionné par le NIST comme successeur de DES
- **RC** (**R**ivest **C**ipher) : blocs de taille variable, réputé plus rapide que DES, existe en plusieurs déclinaisons (RC2, RC5, RC4 qui est à la base des chiffrements utilisés dans **SSL** (**S**ecure **S**ockets **L**ayer – celui de **https**) et **WEP** (**W**ired **E**quivalent **P**rivacy) utilisé dans les réseaux Wifi)

Tous ces algorithmes sont **publics**, leur fonctionnement n'est pas un mystère → leur efficacité repose sur la complexité des clés utilisées → ils sont toujours accompagnés d'un **algorithme de génération de clés**

On appelle **clés faibles [weakkeys]** les clés qui conduisent à des chiffrements faciles à déchiffrer.

Les chiffrements asymétriques

Principe général :

Chaque utilisateur U dispose de 2 clés :

1) une **clé publique** : connue de tous (accompagné d'un **certificat**, voir plus loin) → utilisée par tout autre utilisateur pour **crypter** un message destiné à U

2) une **clé privée** : connue de U seul → utilisée par U pour **décrypter** tout message qui a été crypté avec sa clé publique

Dès lors, n'importe qui peut envoyer un message crypté à U mais seul U peut le décrypter.

Comme on le verra plus tard, le **certificat** permet de certifier que la clé publique dont on dispose appartient bien à la personne visée.

De manière générale, les algorithmes asymétriques se basent sur l'**arithmétique modulaire** et les logarithmes discrets.

Exemple :

Message à coder : « hello » (codes ASCII : 104 – 101 – 108 – 108 – 111)

Clé publique du destinataire : **(n,e)** = (3233,17)

Clé privée du destinataire : **(n,d)** = (3233,2753)

Algorithme de cryptage (RSA) : <caractère chiffré> = <caractère clair>^{e} % **n**

Algorithme de décryptage (RSA) : <caractère clair> = <caractère chiffré>^{d} % **n**

Les clés publiques et privées sont donc des couples d'entiers (n,e) et (n,d) :

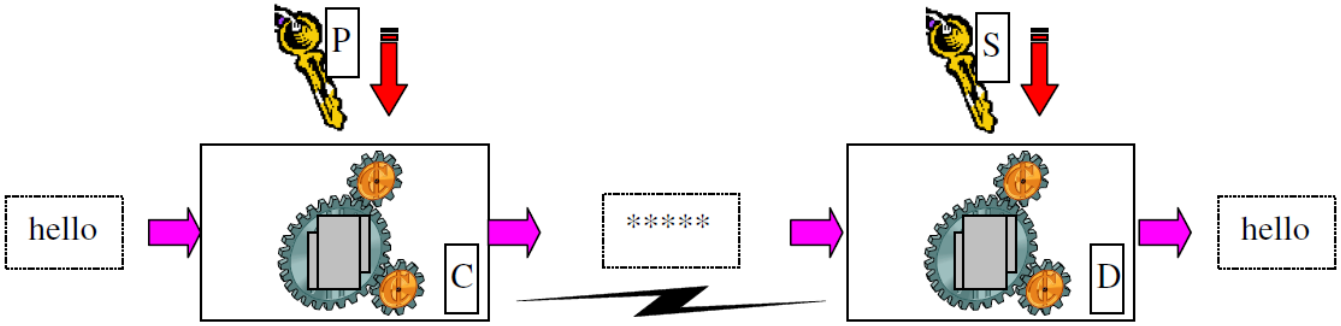
n = module

e = exposant public

d = exposant privé

chiffrement :

message clair	"hello"	104	101	108	108	111
clé publique du destinataire = (3233, 17)						
c^{17} % 3233						
message crypté	*****	2170	1313	745	745	2185



(P=clé Publique; S=clé privée, qui doit rester Secrète)

déchiffrement :

message crypté	*****	2170	1313	745	745	2185
clé **privée** du destinataire = (3233, 2753)						
c^{2753} % 3233						
message obtenu	"hello"	104	101	108	108	111

Comment les clés publiques/privées sont-elles construites ?

1) On choisit aléatoirement deux **nombres premiers p** et **q** relativement grand → ici p=61 et q=53)
2) Le module est alors égal à **n = p*q** → ici, n = 61*53 = 3233
3) **e** est un entier choisi dans [3 , n-1] et premier avec z = (p-1)*(q-1) → ici, e = 17 est premier avec z = 60*52 = 3120 (pas de facteur commun)
4) **d** est choisi tel que e*d-1 soit divisible par z → ici d = 2753 et 17*2753-1 = 46800 est divisible par 3120

Quelques algorithmes courants :

- **RSA** (du nom de ses inventeurs **R**ivest, **S**hamir et **A**dleman) : algorithme le plus utilisé au monde, considéré comme efficace avec des clés de 1024 bits (voir exemple ci-dessus)
- **EIGamal**
- **LUC**

Utilisation correcte des chiffrements symétriques : les **clés de session**

Personne n'utilise un chiffrement asymétrique pour coder une série de messages !

→ Le temps et les ressources nécessaires sont trop importants

→ En pratique, on utilise une **approche hybride**, combinant chiffrement **asymétrique** et **symétrique** : la **clé de session**

Supposons que l'utilisateur **A** veuille communiquer de manière codée avec l'utilisateur **B**.

Dans un premier temps :

1) **A** génère une **clé secrète** (pour un algorithme symétrique) : cette clé est appelée de la **clé de session**

2) **A crypte asymétriquement la clé de session avec la clé publique de B**

3) **A** envoie la clé de session cryptée à **B**

4) **B décrypte la clé de session cryptée avec sa clé privée**

5) **A** et **B** dispose alors tous les deux de la même **clé de session**

Cette procédure porte le nom de « **procédure d'échange de clé** » ou encore « **procédure de handshake** ».

Dans un second temps, A et B peuvent d'échanger des messages cryptés symétriquement à l'aide de la **clé de session** commune.

Il existe une variante à cette procédure qui permettrait à A et B de générer localement la même **clé de session** sur base de renseignements échangés sur le réseau (exemple : algorithme de Diffie et Hellman)

Exemple (**Diffie et Hellman**) :

Supposons que les utilisateurs **A** et **B** veulent communiquer de manière codée. Alors

1) **A** et **B** se mettent d'accord sur deux nombres **n** et **p** (n < p) → **p=11** et **n=7** → paramètres d'une fonction puissance en arithmétique modulaire : $\mathbf{n^x \% p = 7^x \% 11}$
2) **A** choisit aléatoirement un nombre **a** qu'il garde secret → **a=3**
3) **B** choisit aléatoirement un nombre **b** qu'il garde secret → **b=6**
4) **A** calcule $\mathbf{\alpha = 7^a \% 11} = 7^3 \% 11 = 343 \% 11 \mathbf{= 2}$ → sa **clé publique** qu'il envoie à **B**
5) **B** calcule $\mathbf{\beta = 7^b \% 11} = 7^6 \% 11 = 117649 \% 11 \mathbf{= 4}$ → sa **clé publique** qu'il envoie à **A**
6) **A** construit une **clé secrète** à l'aide de la clé publique de **B** :
   $\mathbf{clé_A = \beta^a \% 11} = 4^3 \% 11 \mathbf{= 9}$
7) **B** construit une clé secrète à l'aide de la clé publique de **A** :
   $\mathbf{clé_B = \alpha^b \% 11} = 2^6 \% 11 \mathbf{= 9}$
8) **A** et **B** ont la même clé secrète !!! → ils peuvent communiquer par cryptage symétrique

Comment peut-on être sûr qu'un hacker ne pourra pas lui aussi construire la même clé avec les éléments publics échangés ? Parce qu'il faut connaître **a** ou **b** qui sont restés secrets et qui son très difficile à deviner (la fonction % n'est pas réversible).

Les classes JAVA de cryptographie : interface et implémentation

L'interface de référence

Le **JCA** (**J**ava **C**ryptography **A**rchitecture) fournit les classes et interfaces de base des concepts cryptographiques :

- L'interface **Key** (représentant une clé) (**package java.security** du JDK) avec les méthodes :
  - public abstract String **getAlgorithm**()
  - public abstract byte[] **getEncoded**()
  - public abstract String **getFormat**()

- La classe **Cipher** (dont un objet permettra de crypter un message) (**package javax.crypto** du JDK) avec les méthodes :
  - public final byte[] **crypt**(byte[] in)
  - public final String **getAlgorithm**()
  - public **static Cipher getInstance**(String algorithm)
  - …

Le **JCE** (**J**ava **C**ryptography **E**xtension) est une extension du JCA qui

- comporte l'implémentation de l'encryptage et l'échange de clés
- est séparé du JCA parce que les USA considèrent qu'il s'agit d'une arme → il en limite l'usage aux USA et Canada

Dès lors, les classes réellement utilisées et instanciées ne sont pas connues à l'avance et devront être fournies par des tiers. Le JCA propose donc d'utiliser la démarche suivante :

- Les **classes cryptographiques** ne sont pas instanciées par des constructeurs nécessitant de connaître le nom de ces classes
- Les **classes cryptographiques** seront instanciées à l'aide de **méthodes factory**, et donc leur type sera déterminé au moment de l'exécution et non lors de la compilation → cf. méthode **getInstance** de la classe **Cipher**

L'implémentation et les providers

On appelle **CSP** (**C**ryptographic **S**ervice **P**rovider) ou simplement « **providers** » les sociétés / organisations de développeurs qui fournissent une implémentation du JCE. On peut citer

- **Sun** (Oracle aujourd'hui) qui a donc développé son propre JCE mais dont l'usage est limité aux USA et Canada
- **Cryptix** : simple librairie exemple restée inchangée depuis 2005
- **Bouncy Castle** : librairie renommée et très utilisée, libre open-source → l'ensemble des contributeurs à cette librairie se désigne par le nom de « Legion of the Bouncy Castle » (http://www.bouncycastle.org) → provider pour le JCA et le JCE

A chaque provider correspond un objet instanciant une classe dérivée de la classe **Provider** du JDK, classe qui hérite elle-même de la classe **Properties** (dont les propriétés sont les algorithmes effectivement implémentés).

Lors de l'installation du JDK, une **liste de providers** est fournie **par défaut**. Ceux-ci peuvent être récupéré par l'intermédiaire de la classe **Security** qui accède au fichier **java.security** qui se trouve dans le répertoire d'installation du JRE :

```
# pwd
/usr/local/jdk1.8.0_111/jre/lib/security
# cat java.security
#
# This is the "master security properties file".
#
...
# In this file, various security properties are set for use by
# java.security classes. This is where users can statically register
# Cryptography Package Providers ("providers" for short). The term
# "provider" refers to a package or set of packages that supply a
# concrete implementation of a subset of the cryptography aspects of
# the Java Security API. A provider may, for example, implement one or
# more digital signature algorithms or message digest algorithms.
#
# Each provider must implement a subclass of the Provider class.
# To register a provider in this master security properties file,
# specify the Provider subclass name and priority in the format
#
#    security.provider.<n>=<className>
#
# This declares a provider, and specifies its preference
# order n. The preference order is the order in which providers are
# searched for requested algorithms (when no specific provider is
# requested). The order is 1-based; 1 is the most preferred, followed
# by 2, and so on.
```

```
#
# <className> must specify the subclass of the Provider class whose
# constructor sets the values of various properties that are required
# for the Java Security API to look up the algorithms or other
# facilities implemented by the provider.
#
# There must be at least one provider specification in java.security.
# There is a default provider that comes standard with the JDK. It
# is called the "SUN" provider, and its Provider subclass
# named Sun appears in the sun.security.provider package. Thus, the
# "SUN" provider is registered via the following:
#
#    security.provider.1=sun.security.provider.Sun
#
# (The number 1 is used for the default provider.)
#
# Note: Providers can be dynamically registered instead by calls to
# either the addProvider or insertProviderAt method in the Security
# class.

#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=sun.security.rsa.SunRsaSign
security.provider.3=sun.security.ec.SunEC
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
security.provider.5=com.sun.crypto.provider.SunJCE
security.provider.6=sun.security.jgss.SunProvider
security.provider.7=com.sun.security.sasl.Provider
security.provider.8=org.jcp.xml.dsig.internal.dom.XMLDSigRI
security.provider.9=sun.security.smartcardio.SunPCSC

#
# Sun Provider SecureRandom seed source.
#
...
#
```

Lorsque qu'une **méthode factory**, qui est toujours une méthode de classe, réclame un objet précis implémentant un algorithme donné, la classe correspondante va rechercher, par le biais de la classe **Security**, la liste des providers enregistrés et, selon l'ordre prédéfini, va leur demander un objet de la classe algorithme recherché.

La classe **Security** possède la méthode

- public static Provider[] **getProviders**()

qui permet de récupérer cette liste, tandis que la classe **Provider** possède les méthodes

- public String **getName**()

- public double **getVersion**()
- public void **list**(PrintStreamout) héritée de la classe **Properties**

Exempe : ListProviders.java :

```
import java.security.*;

public class ListProviders
{
    public static void main(String args[])
    {
        Provider prov[] = Security.getProviders();
        for (int i=0; i<prov.length; i++)
            System.out.println(prov[i].getName() + "/" + prov[i].getVersion());
    }
}
```

dont un exemple d'exécution fournit

```
# java ListProviders
SUN/1.8
SunRsaSign/1.8
SunEC/1.8
SunJSSE/1.8
SunJCE/1.8
SunJGSS/1.8
SunSASL/1.8
XMLDSig/1.8
SunPCSC/1.8
#
```

Ajouter et enregistrer des providers complémentaires

Il est tout d'abord nécessaire de se munir du **jar** correspondant à l'implémentation du provider. Pour ce qui est du provider « **Bouncy Castle** » que nous allons utiliser ici, il s'agit de **bcprov-jdk15to18-169.jar** (pour le JDK allant de 15 à 18).

Pour l'utiliser, 2 méthodes possibles :

1) De manière statique :
    a. On place le **jar** dans le répertoire **jre/lib/ext** du JRE, ce qui en fait une librairie d'extension définitive du JRE
    b. On modifie, si on dispose des droits dessus, le fichier **java.security** en ajoutant **Bouncy Castle** dans la liste des providers :

```
security.provider.10=org.bouncycastle.jce.provider.BouncyCastleProvider
```

2) De manière dynamique :

a. On monte le **jar** dans le projet Java en cours

b. On utilise la méthode

public static int **addProvider**(Provider provider)

de la classe **Security**. Dans le cas de Bouncy Castle, cela correspond à

```
Security.addProvider(new BouncyCastleProvider());
```

Notre exemple (ListProviders.java) devient alors

```
import java.security.*;
//import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class ListProviders
{
    public static void main(String args[])
    {
        //Security.addProvider(new BouncyCastleProvider());

        Provider prov[] = Security.getProviders();
        for (int i=0; i<prov.length; i++)
            System.out.println(prov[i].getName() + "/" + prov[i].getVersion());
    }
}
```

dont l'exécution fournit à présent :

```
# java ListProviders
SUN/1.8
SunRsaSign/1.8
SunEC/1.8
SunJSSE/1.8
SunJCE/1.8
SunJGSS/1.8
SunSASL/1.8
XMLDSig/1.8
SunPCSC/1.8
BC/1.69
#
```

Si le fichier **java.security** n'a pas été modifié, il est nécessaire de décommenter les 2 lignes du code ci-dessus.

Cryptage symétrique en pratique

Un générateur de clé

Avant toute chose, il est nécessaire de générer une clé secrète. Pour cela, on dispose de la classe **KeyGenerator** qui possède les méthodes

- public static KeyGenerator **getInstance**(String algorithm,String provider)
- public static KeyGenerator **getInstance**(String algorithm)

Dans le 1er cas, le nom du provider est précisé alors que dans le 2ème cas, le provider sera choisi automatiquement en suivant l'ordre des providers dicté par le fichier **java.security**.

Par exemple, si on veut obtenir un générateur de clé pour un algorithme DES en utilisant le provider Bouncy Castle, nous aurons

```
KeyGenerator cleGen = KeyGenerator.getInstance("DES","BC");
```

L'objet obtenu va nous permettre de générer une clé. Il faut tout d'abord l'initialiser de manière aléatoire à l'aide de sa méthode

- public void **initialize**(SecureRandom random)

où **SecureRandom** est une classe dérivée de la classe **Random**. Nous aurons alors

```
cleGen.initialize(new SecureRandom());
```

Une clé secrète pour l'algorithme visé peut enfin être obtenue à l'aide de la méthode

- public **SecretKey** **generateKey**()

où **SecretKey** est une interface qui hérite de l'interface **Key**. Nous aurons donc

```
SecretKey cle = clenGen.generateKey();
```

Chiffrer/déchiffrer un message

Avant toute chose, nous avons besoin d'un objet représentant l'algorithme de chiffrement. Pour cela, on dispose de la classe **Cipher** qui possède les méthodes

- public static Cipher **getInstance**(String algorithm,String provider)
- public static Cipher **getInstance**(String algorithm)

où, dans la chaîne de caractères « algorithm », il faut préciser

- le **nom de l'algorithme**
- le **mode de chiffrement**
- le **type de padding**

séparés par le caractère « **/** ».

Par exemple, si on veut obtenir, via le provider Bouncy Castle, un objet destiné à crypter/décrypter des données avec l'algorithme DES, le mode de chiffrement ECB et un padding du type PKCS#5, nous aurons

```
Cipher chiffrement = Cipher.getInstance("DES/ECB/PKCS5Padding","BC");
```

Il est ensuite nécessaire d'initialiser cet objet à l'aide de la clé secrète. Pour cela, on utilise la méthode

- public final void **init**(int optmode,Key key)

où le 1er paramètre peut prendre une des valeurs

- public static final int **ENCRYPT_MODE** = 1;
- public static final int **DECRYPT_MODE** = 2;

Dans notre exemple, si on configurer notre objet pour crypter, nous aurons

```
chiffrement.init(Cipher.ENCRYPT_MODE,cle);
```

Le cryptage/décryptage se réalise alors en utilisant la méthode

- public final byte[] **doFinal**(byte[] in)

où la méthode reçoit en paramètre le message à crypter (sous forme d'un tableau de bytes) et retourne le message crypté (sous forme d'un tableau de bytes également).

Dans notre exemple, cela pourrait donner

```
byte[] messageCrypte = chiffrement.doFinal(messageClair) ;
```

Exemple basique avec DES (CryptSymDES.java)

```
import java.security.*;
import javax.crypto.*;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class CryptSymDES
{
    public static void main(String args[]) throws NoSuchAlgorithmException,
NoSuchProviderException, NoSuchPaddingException, InvalidKeyException,
IllegalBlockSizeException, BadPaddingException
    {
        Security.addProvider(new BouncyCastleProvider());

        // Génération de la clé secrète
        KeyGenerator cleGen = KeyGenerator.getInstance("DES","BC");
        cleGen.init(new SecureRandom());
        SecretKey cle = cleGen.generateKey();
        System.out.println("***** Clé générée = " + cle.toString());

        // Chiffrement
        Cipher chiffrementE = Cipher.getInstance("DES/ECB/PKCS5Padding","BC");
        chiffrementE.init(Cipher.ENCRYPT_MODE, cle);

        byte[] texteClair = "Quel beau message que voila".getBytes();
        byte[] texteCrypte = chiffrementE.doFinal(texteClair);

        System.out.println("Cryptage   : " + new String(texteClair) + " ---> " + new
String (texteCrypte));

        // Déchiffrement
        Cipher chiffrementD = Cipher.getInstance("DES/ECB/PKCS5Padding","BC");
        chiffrementD.init(Cipher.DECRYPT_MODE,cle);
        byte[] texteDecrypte = chiffrementD.doFinal(texteCrypte);
        System.out.println("Decryptage : " + new String(texteCrypte) + " ---> " + new
String(texteDecrypte));
    }
}
```

dont un exemple d'exécution fournit :

```
# java CryptSymDES
***** Clé générée = javax.crypto.spec.SecretKeySpec@fffe7843
Cryptage   : Quel beau message que voila ---> "��  PFr Z #  9a pc e G /l
Decryptage : "��  PFr Z #  9a pc e G /l ---> Quel beau message que voila
#
```

On peut remarquer qu'un ensemble d'exceptions sont susceptibles d'être lancées. Il faudrait donc les traiter proprement à l'aide de **try…catch**.

Exemple basique avec AES (Rijndael) (CryptSymAES.java)

```
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.IvParameterSpec;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class CryptSymAES
{
    public static void main(String args[]) throws NoSuchAlgorithmException, …
    {
        Security.addProvider(new BouncyCastleProvider());

        // Génération de la clé secrète
        KeyGenerator cleGen = KeyGenerator.getInstance("Rijndael","BC");
        cleGen.init(128, new SecureRandom());
        SecretKey cle = cleGen.generateKey();
        System.out.println("***** Clé générée = " + cle.toString());

        // Chiffrement
        Cipher chiffrement = Cipher.getInstance("Rijndael/CBC/PKCS5Padding","BC");
        byte[] vecteurInit = new byte[16];
        SecureRandom sr = new SecureRandom();
        sr.nextBytes(vecteurInit);
        chiffrement.init(Cipher.ENCRYPT_MODE,cle,new IvParameterSpec(vecteurInit));

        byte[] texteClair = "Ceci est un message a chiffrer".getBytes();
        byte[] texteCrypte = chiffrement.doFinal(texteClair);
        System.out.println("Cryptage   : " + new String(texteClair) + " ---> " + new
String(texteCrypte));

        // Déchiffrement
        chiffrement.init(Cipher.DECRYPT_MODE,cle,new IvParameterSpec(vecteurInit));
        byte[] texteDecrypte = chiffrement.doFinal(texteCrypte);
        System.out.println("Decryptage : " + new String(texteCrypte) + " ---> " + new
String(texteDecrypte));
    }
}
```

dont un exemple d'exécution fournit :

```
# java CryptSymAES
***** Clé générée = javax.crypto.spec.SecretKeySpec@afd30a97
Cryptage   : Ceci est un message a chiffrer ---> V7gh<OLy@ælTN*)
N}
Decryptage : V7gh<OLy@ælTN*) N} ---> Ceci est un message a
chiffrer
#
```

On constate que :

- Contrairement à **DES** qui utilise un système de blocs indépendants (**EBC**), **Rijndael** (plus connu sous le nom **AES**) utilise un **système de blocs chaînés** (**CBC**). Pour rappel, chaque bloc de texte clair est combiné avec le bloc de texte chiffré précédent, ce qui implique l'existence d'un **vecteur d'initialisation (IV)** afin de permettre le cryptage du premier bloc.
- L'object **cleGen** est initialisé avec la taille de la clé et une instance de la classe **SecureRandom**.
- L'objet **sr** instanciant la classe **SecureRandom** permet d'initialiser aléatoirement le vecteur de bytes **vecteurInit** qui permet de créer une instance de la classe **IvParameterSpec** représentant IV.
- Les objets instanciant la classe **Cipher** doivent être initialisés avec cet objet qui doit être connu au cryptage et au décryptage. Celui-ci devrait donc être transmis également lors d'une communication.

Exemple DES avec le réseau (Version 1)

Nous allons imaginer un exemple dans lequel :

- Un processus client va envoyer par le réseau un message (dont les données claires sont une chaîne de caractères (**nom**) et un entier (**age**) à un processus serveur en attente sur le port 10000.
- Le message clair sera crypté en **DES** à l'aide d'une clé sécrète. Cette clé secrète sera générée par un programme indépendant qui la stockera dans un fichier sérialisé (**cleSecrete.ser**) que les processus client et serveur devront obtenir d'une manière ou d'une autre (clé USB, …).
- Une classe **MyCrypto** va être créée afin de servir de boîte à outils cryptographiques. Cette classe contiendra des méthodes statiques de cryptage et décryptage.

Commençons par la classe **MyCrypto** (fichier **MyCrypto.java**) :

```
package MyCrypto;

import java.security.*;
import javax.crypto.*;

public class MyCrypto
```

```
{
    public static byte[] CryptSymDES(SecretKey cle,byte[] data) throws …
    {
        Cipher chiffrementE = Cipher.getInstance("DES/ECB/PKCS5Padding","BC");
        chiffrementE.init(Cipher.ENCRYPT_MODE, cle);
        return chiffrementE.doFinal(data);
    }

    public static byte[] DecryptSymDES(SecretKey cle,byte[] data) throws …
    {
        Cipher chiffrementD = Cipher.getInstance("DES/ECB/PKCS5Padding","BC");
        chiffrementD.init(Cipher.DECRYPT_MODE, cle);
        return chiffrementD.doFinal(data);
    }
}
```

Ces 2 méthodes de classe prennent en paramètre la clé secrète et le tableau de bytes à crypter/décrypter et retournent le tableau de bytes cryptés/décryptés.

Programme permettant de générer la clé de secrète (fichier **GenereCleDES.java**) :

```
import java.io.*;
import java.security.*;
import javax.crypto.*;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class GenereCleDES
{
    public static void main(String args[]) throws NoSuchAlgorithmException, …
    {
        Security.addProvider(new BouncyCastleProvider());

        // Génération de la clé secrète
        KeyGenerator cleGen = KeyGenerator.getInstance("DES","BC");
        cleGen.init(new SecureRandom());
        SecretKey cle = cleGen.generateKey();
        System.out.println("***** Clé générée = " + cle.toString());

        // Sérialisation de la clé secrète dans un fichier
        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("cleSecrete.ser"));
        oos.writeObject(cle);
        oos.close();
        System.out.println("Sérialisation de la clé dans le fichier cleSecrete.ser");
    }
}
```

dont un exemple d'exécution fournit :

```
# java GenereCleDES
***** Clé générée = javax.crypto.spec.SecretKeySpec@fffe79d1
Sérialisation de la clé dans le fichier cleSecrete.ser
# ls -l
total 4
-rw-rw-r--. 1 student student 133 27 jan 13:48 cleSecrete.ser
…
```

```
#
```

La requête qui va transiter entre le client et le serveur sera un objet instanciant la classe Requete (fichier **Requete.java**) :

```
import java.io.Serializable;

public class Requete implements Serializable
{
    private byte[] data;

    public void setData(byte[] d) { data = d; }
    public byte[] getData() { return data; }
}
```

Elle comporte simplement comme variable membre un tableau de bytes « data » correspondant aux données cryptées.

Le programme client est alors (fichier **ClientDES.java**) :

```
import MyCrypto.MyCrypto;
import java.io.*;
import java.net.Socket;
import java.security.*;
import javax.crypto.*;

public class ClientDES
{
    public static void main(String args[]) throws IOException, …
    {
        // Données à transmettre
        String nom = args[0];
        int age = Integer.parseInt(args[1]);
        System.out.println("Données claires : Nom=" + nom + " Age=" + age);

        // Constructon du vecteur de bytes du message clair
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeUTF(nom);
        dos.writeInt(age);
        byte[] messageClair = baos.toByteArray();
        System.out.println("Construction du message à envoyer");

        // Recuperation de la clé secrète
        SecretKey cle = RecupereCleSecrete();
        System.out.println("Récupération clé secrète : " + cle);

        // Cryptage du message
        byte[] messageCrypte;
        messageCrypte = MyCrypto.CryptSymDES(cle,messageClair);
        System.out.println("Cryptage du message : " + new String(messageCrypte));

        // Construction de la requête
```

```
        Requete req = new Requete();
        req.setData(messageCrypte);

        // Connection sur le serveur
        Socket socket = new Socket("localhost",10000);
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        oos.writeObject(req);
        oos.close();
        socket.close();
        System.out.println("Envoi de la requête");
    }

    public static SecretKey RecupereCleSecrete() throws …
    {
        // Désérialisation de la clé secrète du fichier
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("cleSecrete.ser"));
        SecretKey cle = (SecretKey) ois.readObject();
        ois.close();
        return cle;
    }
}
```

On constate que

- Les données à crypter (un **String** et un **int**) sont lues en ligne de commande et ont tout d'abord été transformées en un tableau de bytes à l'aide des classes **ByteArrayOutputStream** et **DataOutputStream** afin de générer le message clair.
- La clé secrète est récupérée par la fonction de classe **RecupereCleSecrete()** qui récupère ici l'objet instanciant **SecretKey** dans le fichier **cleSecrete.ser**
- Le cryptage de la donnée claire a été réalisée grâce à la classe **MyCrypto** et le message crypté (un tableau de bytes) a été embarqué dans un objet instanciant la classe **Requete**
- Ce qui passe sur le réseau est bel et bien un objet contenant des données cryptées.

Le programme Serveur est alors (fichier **ServeurDES.java**) :

```
import MyCrypto.MyCrypto;
import java.io.*;
import java.net.*;
import java.security.*;
import javax.crypto.*;

public class ServeurDES
{
    public static void main(String args[]) throws …
    {
        // Attente et réception de la requête
        System.out.println("Attente d'une requête...");
```

```
        ServerSocket socket = new ServerSocket(10000);
        Socket socketService = socket.accept();
        ObjectInputStream ois = new
ObjectInputStream(socketService.getInputStream());
        Requete requete = (Requete) ois.readObject();
        ois.close();
        socketService.close();
        socket.close();

        // Recuperation de la clé secrète
        SecretKey cle = RecupereCleSecrete();
        System.out.println("Récupération clé secrète : " + cle);

        // Décryptage du message
        byte[] messageDecrypte;
        System.out.println("Message reçu = " + new String(requete.getData()));
        messageDecrypte = MyCrypto.DecryptSymDES(cle,requete.getData());
        System.out.println("Decryptage du message...");

        // Récupération des données claires
        ByteArrayInputStream bais = new ByteArrayInputStream(messageDecrypte);
        DataInputStream dis = new DataInputStream(bais);
        String nom = dis.readUTF();
        int age = dis.readInt();

        System.out.println("Données claires :");
        System.out.println("Nom = " + nom);
        System.out.println("Age = " + age);
    }

    public static SecretKey RecupereCleSecrete() throws …
    {
        // Désérialisation de la clé secrète du fichier
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("cleSecrete.ser"));
        SecretKey cle = (SecretKey) ois.readObject();
        ois.close();
        return cle;
    }
}
```

On constate que

- La requête est lue sur le réseau en tant qu'objet instanciant la classe Requête
- La clé secrète est récupérée, comme dans le cas du client, à l'aide de la méthode classe **RecupereCleSecrete()**
- Les données cryptées sont extraites de l'objet **requete** et décryptées à l'aide de la classe **MyCrypto**
- Les données claires (**nom** et **age**) sont récupérées en lisant dans le tableau de bytes clairs à l'aide des classes **ByteArrayInputStream** et **DataInputStream**

Un exemple d'exécution au niveau du client est :

```
# java ClientDES wagner 48
Données claires : Nom=wagner Age=48
Construction du message à envoyer
Récupération clé secrète : javax.crypto.spec.SecretKeySpec@fffe79d1
Cryptage du message : ???o駃?C??M
Envoi de la requête
#
```

tandis qu'au niveau du serveur (qui doit être lancé avant le client) :

```
# java ServeurDES
Attente d'une requête...
Récupération clé secrète : javax.crypto.spec.SecretKeySpec@fffe79d1
Message reçu = ???o駃?C??M
Decryptage du message...
Données claires :
Nom = wagner
Age = 48
#
```

Dans cet exemple, tout le secret repose sur la connaissance de la clé secrète. Pour pouvoir crypter/décrypter des messages, il est nécessaire de disposer du fichier sérialisé **cleSecrete.ser**. Chaque intervenant doit disposer de ce fichier et il existe de réelles possibilités de perte ou vol de ce fichier.

Pour remédier à cela, une solution pourrait de générer localement une clé secrète à base de la connaissance d'une donnée secrète commune à l'émetteur et au récepteur, comme un mot de passe par exemple. La classe **SecretKeySpec** permet de créer une clé secrète à partir d'un message (fourni sous la forme d'un tableau de bytes), et cela indépendamment de tout provider. Son constructeur est

- public **SecretKeySpec**(**byte[] message**,String **algorithm**)

Il suffit alors de coder par exemple :

```
SecretKey cle ;
String motDePasse = …
Cle = new SecretKeySpect(motDePasse.getBytes(),"DES");
```

Cependant, le mot de passe doit avoir une longueur valide (**8 bytes** dans le cas de **DES**) et aucune vérification sur la faiblesse potentielle de la clé n'est réalisée par le constructeur. Les 64 bits de clé créé correspondent en fait aux 8 bytes du mot de passe.

Exemple DES avec le réseau (Version 2)

Nous reprenons l'exemple précédent à la différence que

- La clé secrète n'est pas stockée dans un fichier sérialisé mais générée localement à l'aide d'un **mot de passe** fourni en ligne de commande
- La classe **Requete** embarquant le message crypté est directement liée aux données claires (String **nom** et int **age**) qu'elle contient sous forme cryptée, il est donc logique d'embarquer les algorithmes de cryptage/décryptage dans des méthodes propres à la requête → le cryptage se fait dans le constructeur de **Requete** tandis que le décryptage se fait dans les méthodes getNom() et getAge()

La boîte à outils cryptographiques reste inchangée (classe **MyCrypto**).

La classe **Requete** devient à présent (fichier **Requete.java**) :

```
import MyCrypto.MyCrypto;
import java.io.*;
import java.security.*;
import javax.crypto.*;

public class Requete implements Serializable
{
    private byte[] data;

    public Requete(String nom,int age,SecretKey cle) throws …
    {
        // Constructon du vecteur de bytes du message clair
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeUTF(nom);
        dos.writeInt(age);
        byte[] messageClair = baos.toByteArray();

        // Cryptage du message
        byte[] messageCrypte;
        data = MyCrypto.CryptSymDES(cle,messageClair);
    }
```

```
    public String getNom(SecretKey cle) throws …
    {
        // Décryptage du message
        byte[] messageDecrypte;
        messageDecrypte = MyCrypto.DecryptSymDES(cle,data);

        // Récupération des données claires
        ByteArrayInputStream bais = new ByteArrayInputStream(messageDecrypte);
        DataInputStream dis = new DataInputStream(bais);
        String nom = dis.readUTF();
        return nom;
    }

    public int getAge(SecretKey cle) throws …
    {
        // Décryptage du message
        byte[] messageDecrypte;
        messageDecrypte = MyCrypto.DecryptSymDES(cle,data);

        // Récupération des données claires
        ByteArrayInputStream bais = new ByteArrayInputStream(messageDecrypte);
        DataInputStream dis = new DataInputStream(bais);
        String nom = dis.readUTF();
        int age = dis.readInt();
        return age;
    }
}
```

Le programme client (fichier **ClientDES.java**) devient à présent :

```
import java.io.*;
import java.net.Socket;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.SecretKeySpec;

public class ClientDES
{
    public static void main(String args[]) throws …
    {
        // Données à transmettre
        String nom = args[0];
        int age = Integer.parseInt(args[1]);
        String mdp = args[2];
        if (mdp.getBytes().length != 8)
        {
            System.out.println("Mot de passe invalide !");
            System.exit(0);
        }
        System.out.println("Données claires : Nom=" + nom + " Age=" + age);

        // Recuperation de la clé secrète
        SecretKey cle = RecupereCleSecrete(mdp);
```

```
        System.out.println("Récupération clé secrète : " + new
String(cle.getEncoded()));

        // Construction de la requête cryptée
        Requete requete = new Requete(nom,age,cle);

        // Connection sur le serveur
        Socket socket = new Socket("localhost",10000);
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        oos.writeObject(requete);
        oos.close();
        socket.close();
        System.out.println("Envoi de la requête");
    }

    public static SecretKey RecupereCleSecrete(String motDePasse)
    {
        // Génération de la clé secrète à partir du mot de passe
        SecretKey cle = new SecretKeySpec(motDePasse.getBytes(),"DES");
        return cle;
    }
}
```

tandis que le programme serveur (fichier **ServeurDES.java**) devient :

```
import java.io.*;
import java.net.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.SecretKeySpec;

public class ServeurDES
{
    public static void main(String args[]) throws …
    {
        // Vérification de la taille du mot de passe
        String mdp = args[0];
        if (mdp.getBytes().length != 8)
        {
            System.out.println("Mot de passe invalide !");
            System.exit(0);
        }

        // Attente et réception de la requête
        System.out.println("Attente d'une requête...");
        ServerSocket socket = new ServerSocket(10000);
        Socket socketService = socket.accept();
        ObjectInputStream ois;
        ois = new ObjectInputStream(socketService.getInputStream());
        Requete requete = (Requete) ois.readObject();
        ois.close();
        socketService.close();
        socket.close();
```

```
        // Recuperation de la clé secrète
        SecretKey cle = RecupereCleSecrete(mdp);
        System.out.println("Récupération clé secrète : " + new
String(cle.getEncoded()));

        // Récupération des données claires
        String nom = requete.getNom(cle);
        int age = requete.getAge(cle);

        System.out.println("Données claires :");
        System.out.println("Nom = " + nom);
        System.out.println("Age = " + age);
    }

    public static SecretKey RecupereCleSecrete(String motDePasse)
    {
        // Génération de la clé secrète à partir du mot de passe
        SecretKey cle = new SecretKeySpec(motDePasse.getBytes(),"DES");
        return cle;
    }
}
```

Un exemple d'exécution au niveau du client est :

```
# java ClientDES Wagner 48 abcd1234
Données claires : Nom=Wagner Age=48
Récupération clé secrète : abcd1234
Envoi de la requête
#
```

tandis qu'au niveau du serveur (qui doit être lancé avant le client) :

```
# java ServeurDES abcd1234
Attente d'une requête...
Récupération clé secrète : abcd1234
Données claires :
Nom = Wagner
Age = 48
#
```

On constate que le client et le serveur ne sont plus directement concernés par la manière dont les données sont cryptées ou décryptées. Il se contentent d'envoyer et de recevoir la requête et sont ainsi déchargés de la partie « protocole cryptographique ». Cela permet de rendre le code plus générique.

Cryptage asymétrique en pratique

Comme déjà mentionné,

- Un **cryptage asymétrique** n'est jamais utilisé pour crypter une longue série de messages

- Un **cryptage asymétrique** est utilisé la plupart du temps de manière conjointe avec un **cryptage symétrique**, le cryptage asymétrique permettant l'échange de la clé symétrique entre les intervenants

Mais avant cela, commençons par étudier le cryptage asymétrique simple.

Un générateur de clés

Pour qu'un émetteur E puisse envoyer un message à un récepteur R, 2 clés différentes sont nécessaires

1. E doit disposer de la **clé publique** de **R** → le message clair sera crypté à l'aide de cette clé
2. **R** doit disposer de sa **clé privée** → le message crypté sera décrypté à l'aide cette clé → seul R pourra décrypter un message crypté avec sa clé publique.

Avant toute chose, il est donc nécessaire de générer un couple clé privée/clé publique pour le récepteur R. Pour cela, on dispose de la classe **KeyPairGenerator** qui possède les méthodes

- public static KeyPairGenerator **getInstance**(String algorithm,String provider)
- public static KeyPairGenerator **getInstance**(String algorithm)

Dans le 1er cas, le nom du provider est précisé alors que dans le 2ème cas, le provider sera choisi automatiquement en suivant l'ordre des providers dicté par le fichier **java.security**.

Par exemple, si on veut obtenir un générateur de clés pour un algorithme RSA en utilisant le provider Bouncy Castle, nous aurons

```
KeyPairGenerator cleGen = KeyPairGenerator.getInstance("RSA","BC");
```

L'objet obtenu va nous permettre de générer une paire de clés. Il faut tout d'abord l'initialiser de manière aléatoire à l'aide de sa méthode

- public void **initialize**(int keySize, SecureRandom random)

où le premier paramètre est la taille des clés qui seront générées (en bits) et **SecureRandom** est une classe dérivée de la classe **Random**. Nous aurons alors par exemple

```
cleGen.initialize(1024,new SecureRandom());
```

La paire de clés pour l'algorithme visé peut enfin être obtenue à l'aide de la méthode

- public **KeyPair** **generateKeyPair**()

où **KeyPair** est un objet contenant les 2 clés désirées. Celles-ci peuvent récupérées grâce aux méthodes :

- public **PublicKey** **getPublic**()
- public **PrivateKey** **getPrivate**()

où **PublicKey** et **PrivateKey** sont des interfaces héritant de l'interface **Key**.

Nous aurons donc

```
KeyPair deuxCles = cleGen.generateKeyPair();
PublicKey clePublique = deuxCles.getPublic();
PrivateKey clePrivee = deuxCles.getPrivate();
```

Il est alors nécessaire de fournir à l'émetteur E la **clé publique** du récepteur **R**, et il faut que E soit certain que cette clé appartienne bien à R. Cela pourra se faire de manière correcte à l'aide des **certificats** (voir plus loin).

Chiffrer/déchiffrer un message

Cette étape est tout à fait similaire à celle déjà étudiée pour le cryptage/décryptage symétrique → la différence réside dans le choix de l'**algorithme** et les **clés utilisées**.

Par exemple, si on veut obtenir, via le provider Bouncy Castle, un objet destiné à crypter/décrypter des données avec l'algorithme RSA, le mode de chiffrement ECB et un padding du type PKCS#1, nous aurons

```
Cipher chiffrement = Cipher.getInstance("RSA/ECB/PKCS1Padding","BC");
```

Pour le cryptage, nous pourrions avoir

```
chiffrement.init(Cipher.ENCRYPT_MODE,clePublique);
byte[] messageCrypte = chiffrement.doFinal(messageClair) ;
```

Pour le décryptage, nous pourrions avoir

```
chiffrement.init(Cipher.DECRYPT_MODE,clePrivee);
byte[] messageClair = chiffrement.doFinal(messageCrypte) ;
```

Exemple basique avec RSA (fichier CryptAsymRSA.java)

```
import java.security.*;
import javax.crypto.*;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class CryptAsymRSA
{
    public static void main(String args[]) throws …
    {
        Security.addProvider(new BouncyCastleProvider());

        // Génération des clés
        KeyPairGenerator genCles = KeyPairGenerator.getInstance("RSA","BC");
        genCles.initialize(512,new SecureRandom()); // 512 par exemple
        KeyPair deuxCles = genCles.generateKeyPair();
        PublicKey clePublique = deuxCles.getPublic();
        PrivateKey clePrivee = deuxCles.getPrivate();
        System.out.println("*** Cle publique generee = " + clePublique);
        System.out.println("*** Cle privee generee   = " + clePrivee);

        // Cryptage
        Cipher chiffrementE = Cipher.getInstance("RSA/ECB/PKCS1Padding","BC");
        chiffrementE.init(Cipher.ENCRYPT_MODE,clePublique);

        byte[] texteClair = "Le petit cochon est dans la prairie".getBytes();
        byte[] texteCrypte = chiffrementE.doFinal(texteClair);
        System.out.println(new String(texteClair) + " ---> " + new
String(texteCrypte));

        // Décryptage
        Cipher chiffrementD = Cipher.getInstance("RSA/ECB/PKCS1Padding","BC");
        chiffrementD.init(Cipher.DECRYPT_MODE,clePrivee);
        byte[] texteDecrypte = chiffrementD.doFinal(texteCrypte);
        System.out.println(new String(texteCrypte) + " ---> " + new
String(texteDecrypte));
    }
}
```

dont un exemple d'exécution fournit

```
# java CrypAsymRSA
*** Cle publique generee = RSA Public Key
[60:04:2b:39:93:ec:a5:db:f6:24:ac:b5:b0:d3:2b:bb:92:0b:b6:16],[56:66:d1:a4]
        modulus:
e5a70f3fe244c8ff2bff86b276a74db941ec13c097a132e8003ea67005cc33082ea725c21e9c0a5aeba8a
77ea9b7a869db6bfa7e546e99d2098ea449161bb3d3
public exponent: 10001

*** Cle privee generee   = RSA Private CRT Key
[60:04:2b:39:93:ec:a5:db:f6:24:ac:b5:b0:d3:2b:bb:92:0b:b6:16],[56:66:d1:a4]
             modulus:
e5a70f3fe244c8ff2bff86b276a74db941ec13c097a132e8003ea67005cc33082ea725c21e9c0a5aeba8a
77ea9b7a869db6bfa7e546e99d2098ea449161bb3d3
     public exponent: 10001

Le petit cochon est dans la prairie ---> qI��K��[�}��K%1� ;BaX0JG──────────
────O Ø��c�dб;��C�s-F"d

qI��K��[�}��K%1� ;BaX0JG──────────────────────────────────────────────────
────O Ø��c�dб;��C�s-F"d ---> Le petit cochon est dans la prairie
#
```

Exemple RSA combiné avec DES avec le réseau :

échange d'une clé symétrique par cryptage asymétrique

Nous allons reprendre l'exemple déjà utilisé plus haut :

- Un processus client va envoyer par le réseau un message (dont les données claires sont une chaîne de caractères (**nom**) et un entier (**age**) à un processus serveur en attente sur le port 10000.
- Le message clair sera crypté en **DES** à l'aide d'une clé de session. Cette clé de session sera générée par le client lui-même et sera envoyée au serveur de manière cryptée asymétriquement (**RSA**) en utilisant la clé publique du serveur. Le serveur récupérera cette clé de session en la décryptant à l'aide de sa clé privée.
- Les clés publique et privée seront ici stockées dans des fichiers sérialisés (**clePubliqueServeur.ser** dont dispose le client et **clePriveeServeur.ser** dont dispose le serveur)
- La classe **MyCrypto** va être complétée afin de contenir des méthodes statiques de cryptage et décryptage asymétrique (**RSA**).

Commençons par la classe **MyCrypto** (fichier **MyCrypto.java**) :

```
package MyCrypto;

import java.security.*;
import javax.crypto.*;

public class MyCrypto
{
    public static byte[] CryptSymDES(SecretKey cle,byte[] data) throws …
    {
        Cipher chiffrementE = Cipher.getInstance("DES/ECB/PKCS5Padding","BC");
        chiffrementE.init(Cipher.ENCRYPT_MODE, cle);
        return chiffrementE.doFinal(data);
    }

    public static byte[] DecryptSymDES(SecretKey cle,byte[] data) throws …
    {
        Cipher chiffrementD = Cipher.getInstance("DES/ECB/PKCS5Padding","BC");
        chiffrementD.init(Cipher.DECRYPT_MODE, cle);
        return chiffrementD.doFinal(data);
    }

    public static byte[] CryptAsymRSA(PublicKey cle,byte[] data) throws …
    {
        Cipher chiffrementE = Cipher.getInstance("RSA/ECB/PKCS1Padding","BC");
        chiffrementE.init(Cipher.ENCRYPT_MODE, cle);
        return chiffrementE.doFinal(data);
    }

    public static byte[] DecryptAsymRSA(PrivateKey cle,byte[] data) throws …
    {
        Cipher chiffrementD = Cipher.getInstance("RSA/ECB/PKCS1Padding","BC");
        chiffrementD.init(Cipher.DECRYPT_MODE, cle);
        return chiffrementD.doFinal(data);
    }
}
```

Les 2 méthodes de classe ajoutées prennent en paramètre la clé publique/privée du destinataire et le tableau de bytes à crypter/décrypter et retournent le tableau de bytes cryptés/décryptés.

Programme permettant de générer les clés publique/privée du serveur (fichier **GenereClesRSA.java**) :

```
import java.io.*;
import java.security.*;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class GenereClesRSA
{
    public static void main(String args[]) throws …
    {
        Security.addProvider(new BouncyCastleProvider());
```

```
        // Génération des clés
        KeyPairGenerator genCles = KeyPairGenerator.getInstance("RSA","BC");
        genCles.initialize(512,new SecureRandom()); // 512 par exemple
        KeyPair deuxCles = genCles.generateKeyPair();
        PublicKey clePublique = deuxCles.getPublic();
        PrivateKey clePrivee = deuxCles.getPrivate();
        System.out.println(" *** Cle publique generee = " + clePublique);
        System.out.println(" *** Cle privee generee   = " + clePrivee);

        // Sérialisation des clés dans des fichiers différents
        ObjectOutputStream oos1 = new ObjectOutputStream(new
FileOutputStream("clePubliqueServeur.ser"));
        oos1.writeObject(clePublique);
        oos1.close();
        System.out.println("Sérialisation de la clé publique dans le fichier
clePubliqueServeur.ser");

        ObjectOutputStream oos2 = new ObjectOutputStream(new
FileOutputStream("clePriveeServeur.ser"));
        oos2.writeObject(clePrivee);
        oos2.close();
        System.out.println("Sérialisation de la clé privée dans le fichier
clePriveeServeur.ser");
    }
}
```

dont un exemple d'exécution fournit :

```
# java GenereClesRSA
 *** Cle publique generee = RSA Public Key
[5b:ab:cc:ce:d9:84:85:59:11:3b:6e:9a:b4:33:61:00:c9:7a:02:5f],[56:66:d1:a4]
        modulus:
a979505a41d3ca906477fc89c0bd894bfb49e8024b2e31866270f9bee9854b2ef3d5b56e0d3870c0d5289
872c043f9c4c142d988c6a97aa13e1a890d3a369417
public exponent: 10001

 *** Cle privee generee   = RSA Private CRT Key
[5b:ab:cc:ce:d9:84:85:59:11:3b:6e:9a:b4:33:61:00:c9:7a:02:5f],[56:66:d1:a4]
             modulus:
a979505a41d3ca906477fc89c0bd894bfb49e8024b2e31866270f9bee9854b2ef3d5b56e0d3870c0d5289
872c043f9c4c142d988c6a97aa13e1a890d3a369417
     public exponent: 10001

Sérialisation de la clé publique dans le fichier clePubliqueServeur.ser
Sérialisation de la clé privée dans le fichier clePriveeServeur.ser
# ls -l
total 8
-rw-rw-r--. 1 student student 1144 28 jan 09:37 clePriveeServeur.ser
-rw-rw-r--. 1 student student  443 28 jan 09:37 clePubliqueServeur.ser
…
#
```

La requête qui va transiter entre le client et le serveur sera un objet instanciant la classe **Requete** (fichier **Requete.java**) :

```
import java.io.Serializable;

public class Requete implements Serializable
{
    private byte[] data1;  // clé de session cryptée asymétriquement
    private byte[] data2;  // message crypté symétriquement

    public void setData1(byte[] d) { data1 = d; }
    public void setData2(byte[] d) { data2 = d; }
    public byte[] getData1() { return data1; }
    public byte[] getData2() { return data2; }
}
```

Elle comporte comme variables membres

- un tableau de bytes « **data1** » correspondant à la clé de session générée localement et cryptée asymétriquement par le client
- un tableau de bytes « **data2** » correspond aux données cryptées symétriquement par le client en utilisant la clé de session.

Le programme client est alors (fichier **ClientRSA.java**) :

```
import MyCrypto.MyCrypto;
import java.io.*;
import java.net.Socket;
import java.security.*;
import javax.crypto.*;

public class ClientRSA
{
    public static void main(String args[]) throws …
    {
        // Données à transmettre
        String nom = args[0];
        int age = Integer.parseInt(args[1]);
        System.out.println("Données claires : Nom=" + nom + " Age=" + age);

        // Constructon du vecteur de bytes du message clair
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeUTF(nom);
        dos.writeInt(age);
        byte[] messageClair = baos.toByteArray();
        System.out.println("Construction du message à envoyer");

        // Génération d'une clé de session
        KeyGenerator cleGen = KeyGenerator.getInstance("DES","BC");
        cleGen.init(new SecureRandom());
        SecretKey cleSession = cleGen.generateKey();
        System.out.println("Génération d'une clé de session : " + cleSession);
```

```
        // Recuperation de la clé publique du serveur
        PublicKey clePubliqueServeur = RecupereClePubliqueServeur();
        System.out.println("Récupération clé publique du serveur : " +
clePubliqueServeur);

        // Cryptage asymétrique de la clé de session
        byte[] cleSessionCrypte;
        cleSessionCrypte =
MyCrypto.CryptAsymRSA(clePubliqueServeur,cleSession.getEncoded());
        System.out.println("Cryptage asymétrique de la clé de session : " + new
String(cleSessionCrypte));

        // Cryptage symétrique du message
        byte[] messageCrypte;
        messageCrypte = MyCrypto.CryptSymDES(cleSession,messageClair);
        System.out.println("Cryptage symétrique du message : " + new
String(messageCrypte));

        // Construction de la requête
        Requete req = new Requete();
        req.setData1(cleSessionCrypte);
        req.setData2(messageCrypte);

        // Connection sur le serveur
        Socket socket = new Socket("localhost",10000);
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        oos.writeObject(req);
        oos.close();
        socket.close();
        System.out.println("Envoi de la requête");
    }

    public static PublicKey RecupereClePubliqueServeur() throws …
    {
        // Désérialisation de la clé publique
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("clePubliqueServeur.ser"));
        PublicKey cle = (PublicKey) ois.readObject();
        ois.close();
        return cle;
    }
}
```

On constate que

- Les données à crypter (un **String** et un **int**) sont toujours lues en ligne de commande et sont tout d'abord transformées en un tableau de bytes à l'aide des classes **ByteArrayOutputStream** et **DataOutputStream** afin de générer le message clair.
- La clé publique du serveur est récupérée par la fonction de classe **RecupereClePubliqueSeveur()** qui récupère ici l'objet instanciant **PublicKey** dans le fichier **clePubliqueSeveur.ser**

- Le cryptage de la donnée claire et de la clé de session a été réalisé grâce à la classe **MyCrypto** et le résultat de ces 2 cryptages (tableaux de bytes) a été embarqué dans un objet instanciant la classe **Requete**
- Ce qui passe sur le réseau est bel et bien un objet contenant des données et la clé de session cryptées.

Le programme Serveur est alors (fichier **SeveurRSA.java**) :

```
import MyCrypto.MyCrypto;
import java.io.*;
import java.net.*;
import java.security.*;
import javax.crypto.*;
import javax.crypto.spec.SecretKeySpec;

public class ServeurRSA
{
    public static void main(String args[]) throws …
    {
        // Attente et réception de la requête
        System.out.println("Attente d'une requête...");
        ServerSocket socket = new ServerSocket(10000);
        Socket socketService = socket.accept();
        ObjectInputStream ois = new
ObjectInputStream(socketService.getInputStream());
        Requete requete = (Requete) ois.readObject();
        ois.close();
        socketService.close();
        socket.close();

        // Recuperation de la clé privée du serveur
        PrivateKey clePriveeServeur = RecupereClePriveeServeur();
        System.out.println("Récupération clé privée serveur : " + clePriveeServeur);

        // Decryptage asymétrique de la clé de session
        byte[] cleSessionDecryptee;
        System.out.println("Clé session cryptée reçue = " + new
String(requete.getData1()));
        cleSessionDecryptee =
MyCrypto.DecryptAsymRSA(clePriveeServeur,requete.getData1());
        SecretKey cleSession = new SecretKeySpec(cleSessionDecryptee,"DES");
        System.out.println("Decryptage asymétrique de la clé de session...");

        // Décryptage symétrique du message
        byte[] messageDecrypte;
        System.out.println("Message reçu = " + new String(requete.getData2()));
        messageDecrypte = MyCrypto.DecryptSymDES(cleSession,requete.getData2());
        System.out.println("Decryptage symétrique du message...");

        // Récupération des données claires
```

```
        ByteArrayInputStream bais = new ByteArrayInputStream(messageDecrypte);
        DataInputStream dis = new DataInputStream(bais);
        String nom = dis.readUTF();
        int age = dis.readInt();

        System.out.println("Données claires :");
        System.out.println("Nom = " + nom);
        System.out.println("Age = " + age);
    }

    public static PrivateKey RecupereClePriveeServeur() throws …
    {
        // Désérialisation de la clé privée du serveur
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("clePriveeServeur.ser"));
        PrivateKey cle = (PrivateKey) ois.readObject();
        ois.close();
        return cle;
    }
}
```

On constate que

- La requête est lue sur le réseau en tant qu'objet instanciant la classe **Requete**
- La clé privée du serveur est récupérée à l'aide de la méthode classe **RecupereClePriveeServeur()** dans le fichier sérialisé **clePriveeServeur.ser**
- La clé de session est tout d'abord extraite de l'objet **requete** et décryptée à l'aide de la classe **MyCrypto** en utilisant la clé privée du serveur
- Les données cryptées sont extraites de l'objet **requete** et décryptées à l'aide de la classe **MyCrypto** en utilisant la clé de session
- Les données claires (**nom** et **age**) sont récupérées en lisant dans le tableau de bytes clairs à l'aide des classes **ByteArrayInputStream** et **DataInputStream**

Un exemple d'exécution au niveau du client est :

```
# java ClientRSA Wagner 48
Données claires : Nom=Wagner Age=48
Construction du message à envoyer
Génération d'une clé de session : javax.crypto.spec.SecretKeySpec@1845c
Récupération clé publique du serveur : RSA Public Key
[5b:ab:cc:ce:d9:84:85:59:11:3b:6e:9a:b4:33:61:00:c9:7a:02:5f],[56:66:d1:a4]
        modulus:
a979505a41d3ca906477fc89c0bd894bfb49e8024b2e31866270f9bee9854b2ef3d5b56e0d3870c0d5289
872c043f9c4c142d988c6a97aa13e1a890d3a369417
public exponent: 10001

Cryptage asymétrique de la clé de session : 5\��xA��BP_ɪmR�h2�-\�Z��D————————
�4�_��iR¸y'>ţ됩���C��  ►>
```

```
Cryptage symétrique du message : *i���#��UtA|%���
Envoi de la requête
#
```

tandis qu'au niveau du serveur (qui doit être lancé avant le client) :

```
# java ServeurRSA
Attente d'une requête...
Récupération clé privée serveur : RSA Private CRT Key
[5b:ab:cc:ce:d9:84:85:59:11:3b:6e:9a:b4:33:61:00:c9:7a:02:5f],[56:66:d1:a4]
             modulus:
a979505a41d3ca906477fc89c0bd894bfb49e8024b2e31866270f9bee9854b2ef3d5b56e0d3870c0d5289
872c043f9c4c142d988c6a97aa13e1a890d3a369417
     public exponent: 10001

Clé session cryptée reçue =
5\��xA��BP_ɪmR�h2�-\�Z��D──────────────
```

Les message digests

But et principe

Vérifier l'**intégrité** des données transmises → savoir si les données que l'on obtient par le réseau (par exemple) sont restées ce qu'elles étaient à leur envoi

Dans ce but,

- On utilise un **message digest** que l'on joint au message envoyé → il s'agit d'une « **valeur de hachage** » de celui-ci
- Pour créer un message digest, on utilise une **fonction de hachage** qui reçoit un ensemble de données de taille quelconque et fournit une chaîne de taille fixe (typiquement 128 bits)
- Une fonction de hachage idéale n'est pas inversible → il est impossible de retrouver la donnée qui a produit un message digest
- Toutes les données du message sont utilisées pour créer le message digest → la probabilité de trouver 2 message digests identiques pour 2 messages différents est extrêmement faible

Pour vérifier l'intégrité d'une donnée reçue, il faut

1. Calculer un **message digest local** avec le message reçu
2. Comparer le **message digest reçu** avec le message digest créé localement
3. S'ils sont identiques, on peut supposer que le message original n'a pas été altéré

Exemple

Message : « hello »

Fonction de hachage : h(message) = (Somme des codes ASCII des caractères) % 67

On envoie le message clair ainsi que le message digest généré :

message digest d'origine :

message	"hello"	0110 1000	0110 0101	0110 1100	0110 1100	0110 1111
valeur de hachage = h1 = (104+101+108+108+111) % 67 = 63 = *d*						

concaténation

message avec digest	*d*"hello"	63	104	101	108	108	111

(H=Hachage)

message digest à l'arrivée :

message avec digest	*d*"hello"	63	104	101	108	108	111
h1 = 63 h2 = valeur de hachage = (104+101+108+108+111) % 67 = 63							
conclusion							
h1 = h2 → le message n'a pas été modifié							

Quelques algorithmes de message digest courants :

- **MD2**, **MD4** et **MD5** (Rivest) : produisent des digests de 128 bits
- **SHA-1** (**S**ecure **H**ash **A**lgorithm) : plus sûr que MD5, standard du NIST connu sous le nom de **SHS** (**S**ecure **H**ash **S**tandard) → digest de 160 bits pour des blocs d'entrée de 512 bits
- **RIPEMD-160**

Programmer un message digest

Avant toute chose, nous avons besoin d'un objet représentant l'algorithme de hachage. Pour cela, on dispose de la classe **MessageDigest** qui possède la méthode

- public static **MessageDigest** **getInstance**(String algorithm,String provider)

où, dans la chaîne de caractères « algorithm », il faut préciser le nom de l'algorithme de hachage.

Par exemple, si on veut obtenir, via le provider Bouncy Castle, un objet destiné à créer un message digest sur des données avec l'algorithme SHA-1, nous aurons

```
MessageDigest md = MessageDigest.getInstance("SHA-1","BC");
```

Il est ensuite nécessaire de préparer les données sur lesquelles le digest va être calculé. On ajoute ainsi « les ingrédients » en utilisant la méthode

- public void **update**(byte[] input)

qui peut être appelée plusieurs fois s'il y a plusieurs « ingrédients ».

Dans notre exemple, si on veut créer un message digest sur 2 chaînes de caractères « **login** » et « **password** », nous aurons

```
md.update(login.getBytes());
md.update(password.getBytes());
```

Le **message digest** s'obtient alors en utilisant la méthode

- public byte[] **digest**()

Dans notre exemple, cela pourrait donner

```
byte[] digestEnvoye = md.digest() ;
```

Enfin, pour comparer 2 digests, on doit utiliser la méthode

- public static boolean **isEqual**(byte[] digesta,byte[] digestb)

qui compare les deux message digests reçus en paramètre byte par byte et retourne true s'ils sont identiques.

Dans notre exemple, cela pourrait donner

```
boolean test = MessageDigest.isEqual(digestEnvoye,digestLocal);
```

Exemple basique (TestMessageDigest.java)

```
import java.security.*;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class TestMessageDigest
{
    public static void main(String args[]) throws …
    {
        Security.addProvider(new BouncyCastleProvider());

        String login = "wagner", password1 = "abc123", password2 = "abc123";

        System.out.println("Instanciation du message digest 1");
        MessageDigest md1 = MessageDigest.getInstance("SHA-1","BC");
        md1.update(login.getBytes());
        md1.update(password1.getBytes());
        byte[] digest1 = md1.digest();

        System.out.println("Instanciation du message digest 2");
        MessageDigest md2 = MessageDigest.getInstance("SHA-1","BC");
        md2.update(login.getBytes());
        md2.update(password2.getBytes());
        byte[] digest2 = md2.digest();

        System.out.println("Comparaison des digests");
        if (MessageDigest.isEqual(digest1,digest2)) System.out.println("OK !");
        else System.out.println("KO...");
    }
}
```

dont un exemple d'exécution fournit

```
# java TestMessageDigest
Instanciation du message digest 1
Instanciation du message digest 2
Comparaison des digests
OK !
#
```

Utilisation correcte d'un message digest pour une procédure de login

Lors d'une entrée en session sur un serveur, il est courant d'envoyer un couple login/mot de passe. Afin de ne pas envoyer le mot de passe en clair sur le réseau, on pourrait imaginer :

1. créer un **digest** dont les ingrédients sont le login et le mot de passe

2. envoyer le **login** et le **digest** au serveur → le mot de passe ne transite pas en clair sur le réseau
3. le serveur reçoit le **login** (et le **digest**) du client avec lequel il peut retrouver le mot de passe correct (dans une base de données par exemple).
4. Le serveur crée un **digest local** avec le login reçu et le mot de passe récupéré localement.
5. Le serveur **compare** le digest reçu et le digest généré localement → S'ils sont identiques, le client est correctement identifié.

Il existe cependant une **faille importante** à cette méthode

- Le **digest** est construit avec des ingrédients immuables, il est donc toujours identique à chaque nouvelle procédure de login
- Si un hacker intercepte le login et le digest, il peut se faire passer pour le client sans souci dans de futures communications → capturer le digest ou le mot de passe revient au même !

On parle dans ce cas d'un « **digest non salé** ». L'idée est de construire un digest dont les ingrédients changent à chaque nouvelle procédure de login → le digest est donc différent à chaque login et on parle alors de « **digest salé** » → la « **sel** » est dans ce cas les ingrédients qui changent à chaque login et qui sont donc transmis également.

Exemple de procédure de login avec digest salé

Dans cet exemple,

- Un processus client va envoyer une requête de login à un processus serveur en attente sur le port 10000.
- La requête envoyée contiendra le **login**, le **sel** constitué d'un **nombre aléatoire** et de la **date/heure actuelle**, ainsi que le **digest** construit à partir du login, du mot de passe et du sel.
- Sur base du login reçu, le serveur va récupérer le mot de passe local (ici dans une méthode où les logins/mots de passe sont codés en dur → idéalement une base de données par exemple) et créer un **digest local** à partir du login, du sel reçu et du mot de passe local.

La classe **Requete** sera (fichier **Requete.java**) :

```
import java.io.*;
import java.security.*;
import java.util.Date;

public class Requete implements Serializable
{
    private String login;
    private long   temps;
    private double alea;
    private byte[] digest; // digest envoyé

    public Requete(String login,String password) throws …
    {
        this.login = login;

        // Construction du sel
        this.temps = new Date().getTime();
        this.alea  = Math.random();

        // Construction du digest salé
        MessageDigest md = MessageDigest.getInstance("SHA-1","BC");
        md.update(login.getBytes());
        md.update(password.getBytes());
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeLong(temps);
        dos.writeDouble(alea);
        md.update(baos.toByteArray());
        digest = md.digest();
    }

    public String getLogin() { return login; }

    public boolean VerifyPassword(String password) throws …
    {
        // Construction du digest local
        MessageDigest md = MessageDigest.getInstance("SHA-1","BC");
        md.update(login.getBytes());
        md.update(password.getBytes());
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeLong(temps);
        dos.writeDouble(alea);
        md.update(baos.toByteArray());
        byte[] digestLocal = md.digest();

        // Comparaison digest reçu et digest local
        return MessageDigest.isEqual(digest,digestLocal);
    }
}
```

On observe que la classe **Requete**

- embarque dans son **constructeur** (qui sera appelé par le client) l'algorithme de création du **digest salé**
- ne contient pas le mot de passe en clair, mais bien le login, le sel et le digest salé. Les ingrédients du digest ont été ajoutés à l'aide d'un objet instanciant la classe **ByteArrayOutputStream** afin de pouvoir convertir le nombre aléatoire et la date sous forme d'un tableau de bytes
- embarque une méthode **VerifyPassword** (qui sera appelée par le serveur) qui reçoit en paramètre le mot de passe local du client obtenu par le serveur d'une manière ou d'une autre. Cette méthode construit un digest local qui est comparé au digest reçu

Le programme client (fichier **ClientDigest.java**) est :

```
import java.io.*;
import java.net.Socket;
import java.security.*;

public class ClientDigest
{
    public static void main(String args[]) throws …
    {
        // Données à transmettre
        String login = args[0];
        String password = args[1];

        // Construction de la requête de login
        Requete requete = new Requete(login,password);

        // Connexion sur le serveur
        Socket socket = new Socket("localhost",10000);
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        oos.writeObject(requete);
        oos.close();
        socket.close();
        System.out.println("Envoi de la requête");
    }
}
```

tandis que le programme serveur (fichier **ServeurDigest.java**) est :

```
import java.io.*;
import java.net.*;
import java.security.*;

public class ServeurDigest
{
    public static void main(String args[]) throws …
    {
        // Attente et réception de la requête
        System.out.println("Attente d'une requête...");
        ServerSocket socket = new ServerSocket(10000);
        Socket socketService = socket.accept();
        ObjectInputStream ois = new
ObjectInputStream(socketService.getInputStream());
        Requete requete = (Requete) ois.readObject();
        ois.close();
        socketService.close();
        socket.close();

        // Récuperation du mot de passe local
        String motDePasse = RecupereMotDePasse(requete.getLogin());
        if (motDePasse == null)
        {
            System.out.println("Client inconnu !");
            System.exit(0);
        }

        // Vérification du mot de passe reçu
        if (requete.VerifyPassword(motDePasse))
            System.out.println("Bienvenue " + requete.getLogin() + " !");
        else System.out.println("Mauvais mot de passe pour " + requete.getLogin() +
"...");
    }

    public static String RecupereMotDePasse(String login)
    {
        if (login.equals("wagner")) return "abc123";
        if (login.equals("vilvens")) return "azerty";
        return null;
    }
}
```

On constate que le client et le serveur ne sont pas directement concernés par les détails cryptographiques de la procédure de login. Il se contentent d'envoyer et de recevoir la requête et sont ainsi déchargés de la partie « protocole cryptographique ». Cela permet de rendre le code plus générique.

Quelques exemples d'exécution du client sont :

```
# java ClientDigest wagner abc123
Envoi de la requête
# java ClientDigest wagner abc1
Envoi de la requête
# java ClientDigest vilvens xxx
Envoi de la requête
# java ClientDigest charlet 12e
Envoi de la requête
#
```

tandis qu'au niveau du serveur (qui doivent à chaque fois être lancés avant le client) :

```
# java ServeurDigest
Attente d'une requête...
Bienvenue wagner !
# java ServeurDigest
Attente d'une requête...
Mauvais mot de passe pour wagner...
# java ServeurDigest
Attente d'une requête...
Mauvais mot de passe pour vilvens...
# java ServeurDigest
Attente d'une requête...
Client inconnu !
#
```

Les MACs et l'authentification légère

But et principe

- Un message digest permet de vérifier l'**intégrité** d'un message reçu
- Si un hacker intercepte un message ainsi que le digest associé, il peut remplacer les données par d'autres, générer un digest sur ces données corrompues et transmettre le tout au récepteur qui ne se rendra compte de rien → un digest ne permet pas d'assurer l'**authenticité** (**authentification**) du message
- Les **MAC** (**M**essage **A**uthentification **C**ode) peuvent être vus comme des digests permettant de vérifier, en plus de l'intégrité des données, leur authentification.

Il faudrait donc

- Pourvoir créer un MAC à partir des données dont on souhaite assurer l'intégrité mais aussi d'un élément connu uniquement de l'**émetteur** et du **récepteur**
- Dès lors, le MAC est construit à l'aide d'une **clé de session** préalablement échangée entrée l'émetteur et le récepteur → l'**authenticité** est alors assurée également
- Comme pour un digest, un MAC est alors joint aux données claires qui sont envoyées

Deux familles de MAC existent :

1. Le MAC est construit en utilisant une **clé de session** mais pas de fonction de hashage → on parle de MAC simple à clé
2. Le MAC est construit en utilisant une **clé de session** et une **fonction de hashage** → on parle de MAC complexe à clé et hashage → un représentant typique de cette famille est le **HMAC** (keyed-**H**ash **M**essage **A**uthentification **C**ode)

Programmer un HMAC

Il est tout d'abord nécessaire d'instancier un objet de la classe **Mac** qui possède la méthode

- public static final **Mac** **getInstance**(String algorithm,String provider)

où, dans la chaîne de caractères « algorithm », il faut préciser le nom de l'algorithme utilisé.

Par exemple, si on veut obtenir, via le provider Bouncy Castle, un objet destiné à créer un HMAC sur des données avec l'algorithme HMAC-MD5, nous aurons

```
Mac hm = Mac.getInstance("HMAC-MD5","BC");
```

La **clé secrète** (symétrique) est alors fournie à l'objet HMAC au moyen de la méthode

- public final void **init**(Key key)

Il est ensuite nécessaire de préparer les données sur lesquelles le HMAC va être calculé. On ajoute ainsi « les ingrédients » en utilisant la méthode

- public void **update**(byte[] input)

qui peut être appelée plusieurs fois s'il y a plusieurs « ingrédients ».

Dans notre exemple, si on veut utiliser la clé de session « **cleSession** » et créer un HMAC sur 2 chaînes de caractères « **nom** » et « **prenom** », nous aurons

```
hm.init(cleSession);
hm.update(nom.getBytes());
hm.update(prenom.getBytes());
```

Le **HMAC** s'obtient alors en utilisant la méthode

- public final byte[] **doFinal**()

Dans notre exemple, cela pourrait donner

```
byte[] hmac = hm.doFinal() ;
```

Enfin, pour comparer 2 HMAC, on doit toujours utiliser la méthode

- public static boolean **isEqual**(byte[] digesta,byte[] digestb)

de la classe **MessageDigest**.

Dans notre exemple, cela pourrait donner

```
boolean test = MessageDigest.isEqual(hmacEnvoye,hmacLocal);
```

Exemple HMAC avec le réseau

Nous allons imaginer un exemple dans lequel :

- Un processus client va envoyer par le réseau un message (dont les données sont une chaîne de caractères (**nom**) et un entier (**age**) à un processus serveur en attente sur le port 10000.
- Le message sera envoyé en clair mais sera accompagné d'un HMAC des données claires obtenu avec une clé de session, ici obtenue à partir d'un fichier sérialisé. Cette clé de session sera générée par un programme indépendant qui la stockera dans un fichier sérialisé (**cleSession.ser**) que les processus client et serveur devront obtenir d'une manière ou d'une autre (clé USB, …).

Le programme générant la clé de session (DES) est identique à celui déjà rencontré précédemment tandis que la classe **Requete** sera (fichier **Requete.java**) :

```
import java.io.*;
import java.security.*;
import javax.crypto.*;

public class Requete implements Serializable
{
    private String nom;
    private int    age;
    private byte[] hmac; // hmac envoyé

    public Requete(String nom, int age, SecretKey cleSession) throws …
    {
        this.nom = nom;
        this.age = age;

        // Construction du HMAC
        Mac hm = Mac.getInstance("HMAC-MD5","BC");
        hm.init(cleSession);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeUTF(nom);
        dos.writeInt(age);
        hm.update(baos.toByteArray());
        hmac = hm.doFinal();
    }

    public String getNom() { return nom; }
    public int getAge() { return age; }

    public boolean VerifyAuthenticity(SecretKey cleSession) throws …
    {
        // Construction du HMAC local
        Mac hm = Mac.getInstance("HMAC-MD5","BC");
        hm.init(cleSession);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeUTF(nom);
        dos.writeInt(age);
        hm.update(baos.toByteArray());
        byte[] hmacLocal = hm.doFinal();

        // Comparaison HMAC reçu et HMAC local
        return MessageDigest.isEqual(hmac,hmacLocal);
    }
}
```

On observe que la classe **Requete**

- embarque dans son **constructeur** (qui sera appelé par le client) l'algorithme de création du **HMAC** ; il reçoit donc la clé de session en paramètre

- contient bien les données en clair. Les ingrédients du HMAC ont été ajoutés à l'aide d'un objet instanciant la classe **ByteArrayOutputStream** afin de pouvoir convertir la chaîne de caractères et l'entier sous forme d'un tableau de bytes
- embarque une méthode **VerifyAuthenticity** (qui sera appelée par le serveur) qui reçoit en paramètre la clé de session connue localement par le serveur (ici en lisant le fichier sérialisé « **cleSession.ser** »). Cette méthode construit un HMAC local qui est comparé au HMAC reçu

Le programme client (fichier **ClientHMAC.java**) est :

```
import java.io.*;
import java.net.Socket;
import java.security.*;
import javax.crypto.*;

public class ClientHMAC
{
    public static void main(String args[]) throws …
    {
        // Données à transmettre
        String nom = args[0];
        int age = Integer.parseInt(args[1]);
        System.out.println("Données à authentifier : Nom=" + nom + " Age=" + age);

        // Récupération de la clé de session
        SecretKey cle = RecupereCleSession();
        System.out.println("Récupération clé session : " + new
String(cle.getEncoded()));

        // Construction de la requête
        Requete requete = new Requete(nom,age,cle);

        // Connection sur le serveur
        Socket socket = new Socket("localhost",10000);
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        oos.writeObject(requete);
        oos.close();
        socket.close();
        System.out.println("Envoi de la requête");
    }

    public static SecretKey RecupereCleSession() throws …
    {
        // Désérialisation de la clé de session du fichier
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("cleSession.ser"));
        SecretKey cle = (SecretKey) ois.readObject();
        ois.close();
        return cle;
    }
}
```

tandis que le programme serveur (fichier **ServeurHMAC.java**) est :

```
import java.io.*;
import java.net.*;
import java.security.*;
import javax.crypto.*;

public class ServeurHMAC
{
    public static void main(String args[]) throws …
    {
        // Attente et réception de la requête
        System.out.println("Attente d'une requête...");
        ServerSocket socket = new ServerSocket(10000);
        Socket socketService = socket.accept();
        ObjectInputStream ois = new
ObjectInputStream(socketService.getInputStream());
        Requete requete = (Requete) ois.readObject();
        ois.close();
        socketService.close();
        socket.close();

        // Récupération de la clé de session
        SecretKey cle = RecupereCleSession();
        System.out.println("Récupération clé session : " + new
String(cle.getEncoded()));

        // Récupération des données
        String nom = requete.getNom();
        int age = requete.getAge();

        System.out.println("Données à authentifier :");
        System.out.println("Nom = " + nom);
        System.out.println("Age = " + age);
        if (requete.VerifyAuthenticity(cle)) System.out.println("Authentification
validée !");
        else System.out.println("Authentification échouée...");
    }

    public static SecretKey RecupereCleSession() throws …
    {
        // Désérialisation de la clé de session du fichier
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("cleSession.ser"));
        SecretKey cle = (SecretKey) ois.readObject();
        ois.close();
        return cle;
    }
}
```

Un exemple d'exécution du client est :

```
# java ClientHMAC wagner 48
Données à authentifier : Nom=wagner Age=48
Récupération clé session : ms??O?
Envoi de la requête
#
```

tandis qu'au niveau du serveur (qui doit être lancé avant le client) :

```
# java GenereCleDES
***** Clé générée = javax.crypto.spec.SecretKeySpec@fffe787e
Sérialisation de la clé dans le fichier cleSession.ser
# java ServeurHMAC
Attente d'une requête...
Récupération clé session : ms???O?
Données à authentifier :
Nom = wagner
Age = 48
Authentification validée !
#
```

Le fondement de l'authentification est donc la connaissance et le secret de la clé de session.

On parle ici d'« **authentification légère** » car elle ne fait intervenir que des algorithmes peu coûteux en opérations machines (chiffrements symétriques). De plus, tout le mécanisme repose sur le secret de clé de session, ce qui est un inconvénient de la méthode → pourrait-on à la place utiliser des chiffrements asymétriques ? oui ! → les **signatures numériques** → « **authentification lourde** »

Les signatures électroniques et l'authentification lourde

Définition et principe

Une **signature électronique** est un bloc de données qui a été créé en utilisant une clé privée et qui peut être vérifié par la clé publique associée

On parle encore de **signature numérique** ou de **signature digitale**

Sur le principe, la construction d'une signature se fait par

1. Construction d'un message digest sur les données que l'on désire signer
2. Chiffrement asymétrique du digest obtenu à l'aide de la **clé privée du signataire** → on obtient la **signature électronique**

Les données accompagnées de la signature électronique sont alors transmises au destinataire :

1. Celui-ci va déchiffrer asymétriquement la signature reçue avec la **clé publique du signataire** et récupérer le digest créé par le signataire
2. Construire un digest local avec les données reçues et vérifier que celui-ci est identique au digest du signataire. Si c'est le cas :
    a. l'**intégrité** est assurée
    b. l'**authentification** est assurée car cela veut dire que les données ont été signées avec la clé privée du signataire
    c. la **non-répudiation** est assurée pour la même raison : le signataire ne peut pas nier avoir signer ce message

Bien sûr en pratique, la construction et le (dé)chiffrement du digest peuvent se faire en une seule étape à l'aide des classes Java dédiées.

En comparaison d'une signature manuscrite qui ne dépend que du signataire :

Une **signature numérique** dépend du signataire (par l'intermédiaire de sa **clé privée**) mais aussi des **données à signer**

En pratique, un hacker qui aurait dérobé une signature électronique ne peut pas s'en servir pour signer un autre message

Exemple :

Message à signer : « hello » (codes ASCII : 104 – 101 – 108 – 108 – 111)

Fonction de hachage : h(message) = (Σ des codes ASCII des caractères) % 67 = <digest>

Clé publique du signataire « James » : **(n,e)** = (3233,17)

Clé privée du signataire « James » : **(n,d)** = (3233,2753)

Algorithme de cryptage (RSA) : <signature> = <digest>^{e} % **n**

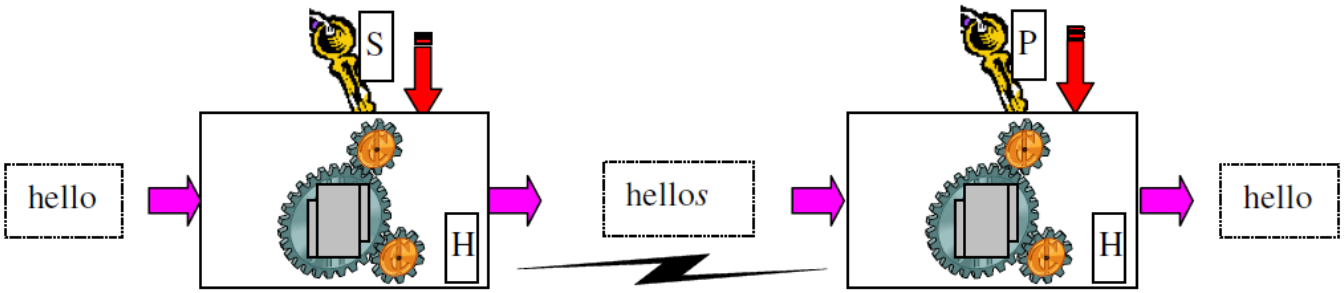
Algorithme de décryptage (RSA) : <digest> = <signature>^{d} % **n**

message digest d'origine :

message	"hello"	0110 1000	0110 0101	0110 1100	0110 1100	0110 1111
valeur de hachage = h1 = (104+101+108+108+111) % 67 = 63						
clé **privée** de James = (n,d) = (3233, 2753)						
signature = s = ($h1^{d}$) % n = (63^{2753}) % 3233 = 1393						

concaténation

message avec digest	"hello"*s*	104	101	108	108	111	1393



message digest à l'arrivée :

message avec digest	"hello"*s*	104	101	108	108	1111	1393
clé publique de James = (3233, 17)							
s = 1393 h1 = (1393^{17}) % 3233 = 63 h2 = valeur de hachage = (104+101+108+108+111) % 67 = 63							
conclusion							
h1 = h2 ➔ le message n'a pas été modifié + c'est bien James qui l'a envoyé							

Dans cet exemple, la **donnée signée** est transmise de manière **claire**. Mais on pourrait imaginer une situation plus complexe où la **donnée signée** est transmise de manière **cryptée** (voir remarque ci-dessous)

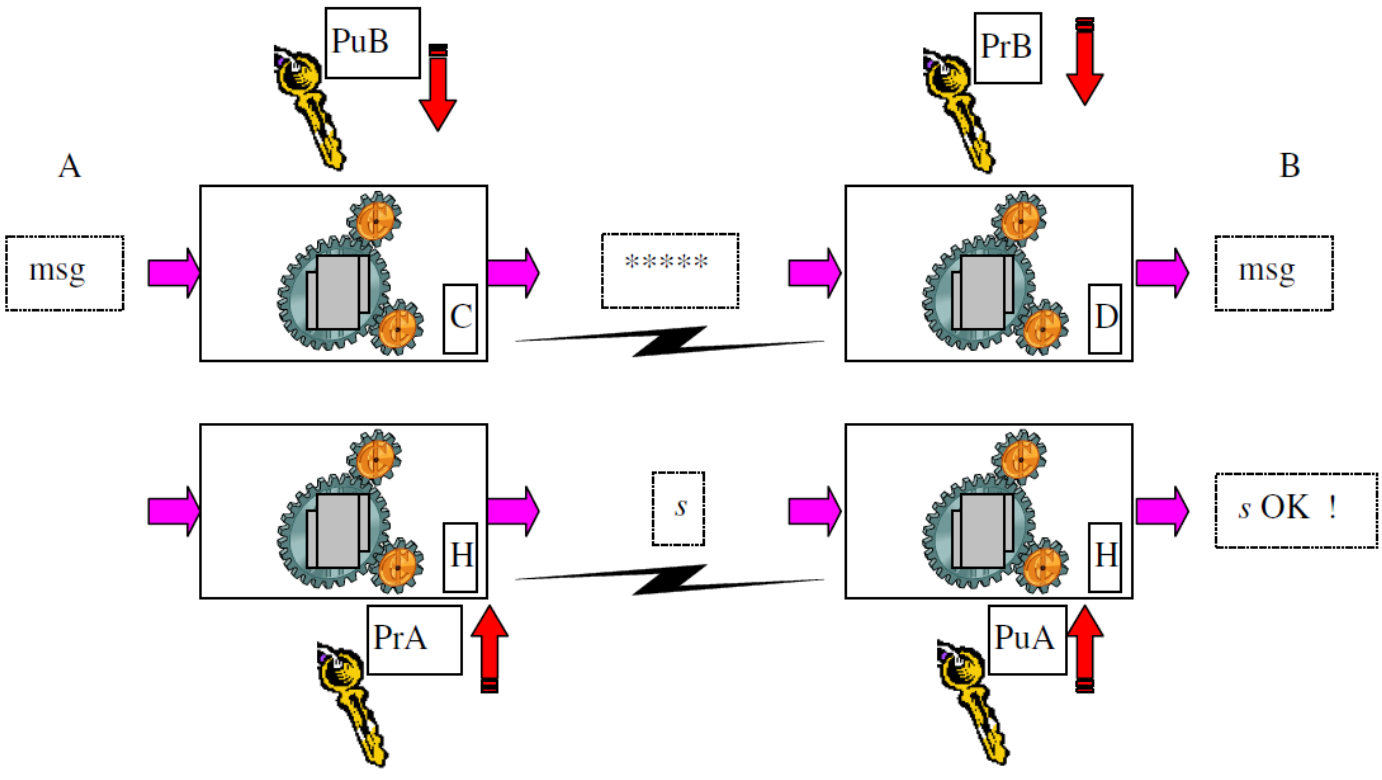
Remarque :

Imaginons deux intervenants A et B dans une situation où

- L'émetteur A veut envoyer un **message crypté** à B et uniquement destiné à B
- B souhaite vérifier l'**intégrité** et l'**authenticité** du message reçu, en s'assurant qu'il vient bien de A.

La séquence des opérations est alors la suivante :

1. A crypte asymétriquement les données claires à l'aide de la **clé publique** de B (ainsi seul B pourra les décrypter)
2. A crée une signature des données claires avec sa **clé privée** (digest des données claires suivi de son cryptage asymétrique à l'aide la clé privée de A)
3. A envoie les données cryptées et sa signature à B
4. B décrypte asymétriquement les données cryptées à l'aide de sa **clé privée**
5. B vérifie la signature reçue en utilisant la **clé publique** de A. Pour cela,
    a. Il décrypte asymétriquement la signature reçue avec la clé publique de A → il récupère le digest des données envoyé par A
    b. Il crée un digest local avec les données claires (décryptées)
    c. Il compare le digest local et le digest reçu → S'il sont identiques, la signature est bien vérifiée



(Pu=clé Publique; Pr=clé privée)

Variante :

On pourrait encore imaginer que les données soient cryptées symétriquement à l'aide d'une **clé de session**, ces données cryptées seraient alors accompagnées d'une **signature** électronique des données claires

Quelques algorithmes de signatures digitales courants :

- **DSA** (**D**igital **S**ignature **A**lgorithm) : utilise une clé de 1024 bits, standard du NIST sous le nom **DSS** (**D**igital **S**ignature **S**tandard)
- **RSA** : algorithme de chiffrement asymétrique (voir exemple ci-dessus) appliqué aux signatures :

  $$\text{signature} = \langle\text{digest}\rangle^{d} \bmod n$$

  où **(n,d)** est la clé privée du signataire

  $$\text{digest} = \langle\text{signature}\rangle^{e} \bmod n$$

  où **(n,e)** est la clé publique correspondante (du signataire donc)

Programmer une signature digitale

Pour le **signataire**, il est tout d'abord nécessaire d'instancier un objet de la classe abstraite (une version dérivée plus précisément) **Signature** qui possède la méthode

- public static **Signature** **getInstance**(String algorithm,String provider)

où, dans la chaîne de caractères « algorithm », il faut préciser le nom de l'algorithme à utiliser.

Par exemple, si on veut obtenir, via le provider Bouncy Castle, un objet destiné à créer une signature sur des données avec l'algorithme SHA1withRSA, nous aurons

```
Signature s = Signature.getInstance("SHA1withRSA","BC");
```

Il faut alors initialiser cet objet avec la **clé privée** du signataire au moyen de la méthode

- public final void **initSign**(**PrivateKey** key)

Il est ensuite nécessaire de préparer les données qui vont être signées. On ajoute ainsi « les ingrédients » en utilisant la méthode

- public final void **update**(byte[] data)

qui peut être appelée plusieurs fois s'il y a plusieurs « ingrédients ».

Dans notre exemple, si on veut utiliser la clé privée « **clePrivee** » pour signer 2 chaînes de caractères « **nom** » et « **prenom** », nous aurons

```
s.initSign(clePriveeSignataire);
s.update(nom.getBytes());
s.update(prenom.getBytes());
```

La signature s'obtient alors en utilisant la méthode

- public final byte[] **sign**()

Dans notre exemple, cela pourrait donner

```
byte[] signature = s.sign() ;
```

On remarque donc qu'il n'est pas nécessaire de créer et crypter un digest explicitement, le tout est réalisé dans la méthode **sign()**.

Pour **vérifier une signature** digitale, le principe reste le même, à la différence que l'on doit initialiser l'objet **Signature** avec la **clé publique** du signataire à l'aide de la méthode

- public final void **initVerify**(**PublicKey** key)

Les « ingrédients » sont ajoutés de la même manière avec la méthode **update()**.

Enfin, la vérification de la signature se réalise à l'aide de la méthode

- public final boolean **verify**(byte[] signature)

de la classe **Signature**.

Dans notre exemple, cela pourrait donner

```
Signature s = Signature.getInstance("SHA1withRSA","BC");
s.initVerify(clePubliqueSignataire);
s.update(nom.getBytes());
s.update(prenom.getBytes());
boolean test = s.verify(signature);
```

Exemple basique (TestSignatureRSA.java)

```
import java.security.*;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

public class TestSignatureRSA
{
    public static void main(String args[]) throws …
    {
        Security.addProvider(new BouncyCastleProvider());

        // Génération des clés
        KeyPairGenerator genCles = KeyPairGenerator.getInstance("RSA","BC");
        genCles.initialize(512,new SecureRandom()); // 512 par exemple
        KeyPair deuxCles = genCles.generateKeyPair();
        PublicKey clePublique = deuxCles.getPublic();
        PrivateKey clePrivee = deuxCles.getPrivate();
        System.out.println(" *** Cle publique generee = " + clePublique);
        System.out.println(" *** Cle privee generee   = " + clePrivee);

        // Données à signer
        String nom = "Wagner";
        String prenom = "Jean-Marc";

        // Creation de la signature
        System.out.println("Création de la signature...");
        Signature s1 = Signature.getInstance("SHA1withRSA","BC");
        s1.initSign(clePrivee);
        s1.update(nom.getBytes());
        s1.update(prenom.getBytes());
        byte[] signature = s1.sign();
        System.out.println("Signature = " + new String(signature));

        // Vérification de la signature
        System.out.println("Vérification de la signature...");
        Signature s2 = Signature.getInstance("SHA1withRSA","BC");
        s2.initVerify(clePublique);
        s2.update(nom.getBytes());
        s2.update(prenom.getBytes());
        boolean test = s2.verify(signature);
        if (test) System.out.println("Signature validée !");
        else System.out.println("Signature invalide...");
    }
}
```

dont un exemple d'exécution est

```
# java Signature.TestSignatureRSA
 *** Cle publique generee = RSA Public Key
[2a:55:c0:c7:6c:bf:ca:97:d5:f7:49:83:11:5e:d9:33:e9:22:3a:ad],[56:66:d1:a4]
        modulus:
b09c2033018e2d503baa92243abc039a769858cd6d3f499d54051757baed04b2d5f177f37ee1393477247
6a7855de0d6b5ac9e5ab7a2c310f85079dd29b8c791
public exponent: 10001

 *** Cle privee generee   = RSA Private CRT Key
[2a:55:c0:c7:6c:bf:ca:97:d5:f7:49:83:11:5e:d9:33:e9:22:3a:ad],[56:66:d1:a4]
             modulus:
b09c2033018e2d503baa92243abc039a769858cd6d3f499d54051757baed04b2d5f177f37ee1393477247
6a7855de0d6b5ac9e5ab7a2c310f85079dd29b8c791
     public exponent: 10001

Création de la signature...
Signature = $eS???x?i???V?gWH?RiJY?Y??x?v!?d??[#?'???_i弊??,I?
Vérification de la signature...
Signature validée !
#
```

Exemple de Signature RSA avec le réseau

Nous allons imaginer un exemple dans lequel :

- Un processus client va envoyer par le réseau un message (dont les données sont une chaîne de caractères (**nom**) et un entier (**age**) à un processus serveur en attente sur le port 10000.
- Le message sera envoyé en clair mais sera accompagné d'une **signature** des données claires obtenue avec la clé privée du client, ici obtenue à partir d'un fichier sérialisé. Cette clé privée sera générée conjointement à la clé publique associée par un programme indépendant qui les stockera dans des fichiers sérialisés (**clePriveeClient.ser** et **clePubliqueClient.ser**). Le processus serveur devra obtenir la clé publique du client d'une manière ou d'une autre (clé USB, … → certificat ; voir plus tard).

Le programme générant les clés privée et publique (RSA) du client est similaire à celui rencontré lors de l'exemple sur le cryptage asymétrique tandis que la classe **Requete** sera (fichier **Requete.java**) :

```java
import java.io.*;
import java.security.*;

public class Requete implements Serializable
{
    private String nom;
    private int    age;
    private byte[] signature; // signature envoyée

    public Requete(String nom,int age,PrivateKey clePriveeClient) throws …
    {
        this.nom = nom;
        this.age = age;

        // Construction de la signature
        Signature s = Signature.getInstance("SHA1withRSA","BC");
        s.initSign(clePriveeClient);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeUTF(nom);
        dos.writeInt(age);
        s.update(baos.toByteArray());
        signature = s.sign();
    }

    public String getNom() { return nom; }
    public int getAge() { return age; }

    public boolean VerifySignature(PublicKey clePubliqueClient) throws …
    {
        // Construction de l'objet Signature
        Signature s = Signature.getInstance("SHA1withRSA","BC");
        s.initVerify(clePubliqueClient);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeUTF(nom);
        dos.writeInt(age);
        s.update(baos.toByteArray());

        // Vérification de la signature reçue
        return s.verify(signature);
    }
}
```

On observe que la classe **Requete**

- embarque dans son **constructeur** (qui sera appelé par le client) l'algorithme de création de la **signature** ; il reçoit donc la clé privée du client en paramètre
- contient bien les données en clair. Les ingrédients à signer ont été ajoutés à l'aide d'un objet instanciant la classe **ByteArrayOutputStream** afin de pouvoir convertir la chaîne de caractères et l'entier sous forme d'un tableau de bytes

- embarque une méthode **VerifySignature** (qui sera appelée par le serveur) qui reçoit en paramètre la clé publique du client connue localement par le serveur (ici en lisant le fichier sérialisé « **clePubliqueClient.ser** »). Cette méthode construit un objet **Signature** local qui permettra de vérifier la signature digitale reçue

Le programme client (fichier **ClientSignatureRSA.java**) est :

```
import java.io.*;
import java.net.Socket;
import java.security.*;

public class ClientSignatureRSA
{
    public static void main(String args[]) throws …
    {
        // Données à transmettre
        String nom = args[0];
        int age = Integer.parseInt(args[1]);
        System.out.println("Données à signer : Nom=" + nom + " Age=" + age);

        // Recuperation de la clé privée du client
        PrivateKey clePrivee = RecupereClePriveeClient();
        System.out.println("Récupération clé privée client...");

        // Construction de la requête
        Requete requete = new Requete(nom,age,clePrivee);

        // Connection sur le serveur
        Socket socket = new Socket("localhost",10000);
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        oos.writeObject(requete);
        oos.close();
        socket.close();
        System.out.println("Envoi de la requête");
    }

    public static PrivateKey RecupereClePriveeClient() throws …
    {
        // Désérialisation de la clé privée client du fichier
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("clePriveeClient.ser"));
        PrivateKey cle = (PrivateKey) ois.readObject();
        ois.close();
        return cle;
    }
}
```

tandis que le programme serveur (fichier **ServeurSignatureRSA.java**) est :

```
import java.io.*;
import java.net.*;
import java.security.*;

public class ServeurSignatureRSA
{
    public static void main(String args[]) throws …
    {
        // Attente et réception de la requête
        System.out.println("Attente d'une requête...");
        ServerSocket socket = new ServerSocket(10000);
        Socket socketService = socket.accept();
        ObjectInputStream ois = new
ObjectInputStream(socketService.getInputStream());
        Requete requete = (Requete) ois.readObject();
        ois.close();
        socketService.close();
        socket.close();

        // Récupération de la clé publique du client
        PublicKey clePublique = RecupereClePubliqueClient();
        System.out.println("Récupération clé publique client...");

        // Récupération des données
        String nom = requete.getNom();
        int age = requete.getAge();

        System.out.println("Données à vérifer :");
        System.out.println("Nom = " + nom);
        System.out.println("Age = " + age);
        if (requete.VerifySignature(clePublique)) System.out.println("Signature
validée !");
        else System.out.println("Signature invalide...");
    }

    public static PublicKey RecupereClePubliqueClient() throws …
    {
        // Désérialisation de la clé publique client du fichier
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("clePubliqueClient.ser"));
        PublicKey cle = (PublicKey) ois.readObject();
        ois.close();
        return cle;
    }
}
```

Un exemple d'exécution du client est :

```
# java ClientSignatureRSA wagner 48
Données à signer : Nom=wagner Age=48
Récupération clé privée client...
Envoi de la requête
#
```

tandis qu'au niveau du serveur (qui doit être lancé avant le client) :

```
#java ServeurSignatureRSA
Attente d'une requête...
Récupération clé publique client...
Données à vérifer :
Nom = wagner
Age = 48
Signature validée !
#
```

Le principe même du bon fonctionnement du système de signature digitale est qu'il est nécessaire de disposer de la clé publique du signataire, et surtout d'être certain que cette clé publique appartient bien au bon signataire. Il est donc indispensable de disposer d'une procédure permettant de **récupérer la bonne clé publique d'une manière fiable** → utilisation des **certificats**

Remarque :

On pourrait imaginer d'envoyer des requêtes signées et cryptées simultanément, et donc de combiner plusieurs des techniques vues jusqu'ici. Par exemple, pour une requête cryptée et signée, on pourrait utiliser la classe

```
public class Requete implements Serializable
{
    private byte[] data;       // Données cryptées
    private byte[] signature; // signature envoyée

    …
}
```

Les certificats

Principe

Dans une communication faisant intervenir

- du cryptage asymétrique
- des signatures digitales

la connaissance de la **clé publique** de l'intervenant avec lequel on veut communiquer est nécessaire. Etant publique, on peut se procurer « facilement » cette clé mais

- comment savoir si on obtient bien une vraie clé ?
- comment savoir qu'il s'agit de la clé de l'intervenant visé ?

Idée : utiliser un troisième intervenant, connu de tout le monde et digne de confiance, qui va attester que la clé publique fournie est bien la bonne. Pour cela, ce 3ème intervenant va créer une **signature digitale** créée par ses soins → c'est ce que l'on appelle un **certificat**

Ce 3ème intervenant (appelé le « tiers de confiance ») porte le nom de **Certification Authority** (**CA**)

Pour être plus précis, un **certificat** (**d'authentification**) contient :

- la **clé publique** d'un intervenant
- l'identification de cet intervenant
- l'identification du CA
- la **signature du CA**

Parmi les CA les plus connus à travers le monde, on peut citer :

- Verisign (http://www.verisign.com)
- Amazone
- Microsoft
- Globalsign

Le contenu d'un certificat peut varier → nécessité d'une norme → la plus répandue est **X.509** pour laquelle un certificat contient

- Numéro de version
- Numéro de série
- Identification de l'algorithme de signature
- **Identification du propriétaire** de la clé publique certifiée
- Période de validité
- **Identification du CA**
- Information sur l'algorithme de clé publique
- **Clé publique**
- Extensions diverses
- **Signature digitale de CA** pour les champs ci-dessus

Que se passe-t-il si le propriétaire de la clé publique certifiée perd sa clé privée (ou se la fait voler) ? Dans ce cas,

- Il faut révoquer le certificat
- Les **CA** gèrent une liste des certificats révoqués : **Certificate Revocation List** (**CRL**)
- On peut aussi s'adresser à un serveur OCSP (Online Certification Status Protocol) → un serveur de ce type est chargé de répondre aux demandes de vérification de validité

Tester la validité d'un certificat

Pour tester la validité d'un certificat,

- Il faut vérifier la signature du CA présente dans le certificat, donc
- Il faut connaître la clé publique du CA

Donc on tourne en rond ! Pour obtenir la clé publique d'un intervenant, il faut connaître la clé publique du CA. Mais comment est-on certain d'avoir la bonne clé du bon CA ?

Deux possibilités :

1. Le **CA est bien connu** (autorité mondiale reconnue) → sa clé publique est alors bien connue aussi → la vérification peut se faire sans problème

2. Le **CA est seulement local** (peu ou pas connu) → il faut se procurer le certificat de ce CA local → qui peut lui aussi être certifiée par un autre CA local → … → qui est certifié par un CA bien connu → on parle de « **chaîne de certificats** »

Les **browsers** actuels possèdent en leur sein, sous forme de certificats, toute une série de clés publiques appartenant à des CA connus (et moins connus mais vérifiés).

Il est également possible de télécharger (par l'intermédiaire des browsers) des certificats qui sont alors enregistrés sur disque selon divers formats, dont les plus courants sont :

- **DER** (**D**efinite **E**ncoding **R**ule) : fichiers d'extension **.der**, **.cer**, **.crt**, **.cert**
- **PEM** (**P**rivacy **E**nhanced **M**ail) : format DER avec des élément en plus, fichiers d'extension **.pem**

Tous ces formats respectent la norme **X.509**

Les classes certificats en Java

Un certificat en Java est représenté par la classe abstraite **Certificate** (du package java.security.cert) qui possède la méthode

- **public abstract PublicKey getPublicKey()** → retourne la clé publique contenue dans le certificat
- **public String getType()** → retourne le type du certificat
- **public abstract void verify(PublicKey key) throws …** → reçoit en paramètre la clé publique du CA qui a signé le certificat et vérifie la validité du certificat

De cette classe dérive la classe abstraite **X509Certificate** qui représente un certificat de la norme X.509.

Pour obtenir une instance d'un objet représentant un certificat, on utilise la classe factory **CertificateFactory** qui possède la méthode

- **public static final CertificateFactory getInstance(String type)**

où type spécifie le type (la norme) du certificat que l'on désire obtenir. On obtient alors une représentation mémoire d'un certificat à l'aide de la méthode

- **public final Certificate generateCertificate(InputStream inStream)**

Où inStream est un flux (associé à un fichier sur disque ou le réseau) sur lequel le certificat doit être lu.

L'objet instanciant un **X509Certificate** contient alors les méthodes suivantes :

- **public abstract Date getNotAfter()** → retourne la date limite de fin de validité du certificat
- **public abstract Date getNotBefore()** → retourne la date de limite de début de validité du certificat
- **public abstract byte[] getSignature()** → retourne la signature du CA
- **public abstract Principal getIssuerDN()** → retourne un objet représentant le CA (**Principal** est une classe représentant un intervenant : un client, un serveur, un CA, … ; celle-ci contient une méthode **getName()**)
- **public abstract Principal getSubjectDN()** → retourne un objet représentant le propriétaire de la clé publique
- **public abstract String getSigAlgName()** → retourne le nom de l'algorithme de signature utilisé par le CA
- …

Les initiales **DN** correspondent à un « **distinguished name** » qui décrit une personne de manière normalisée selon les champs :

- **CN** (**C**ommon **N**ame) : le nom
- **O** (**O**rganization) : l'organisme auquel la personne appartient
- **OU** (**O**rganization **U**nit) : le service auquel cette personne appartient
- **L** (**L**ocality) : la ville
- **S** (**S**tate) : l'état au sens américain, la province/département en Europe
- **C** (**C**ountry) : le pays

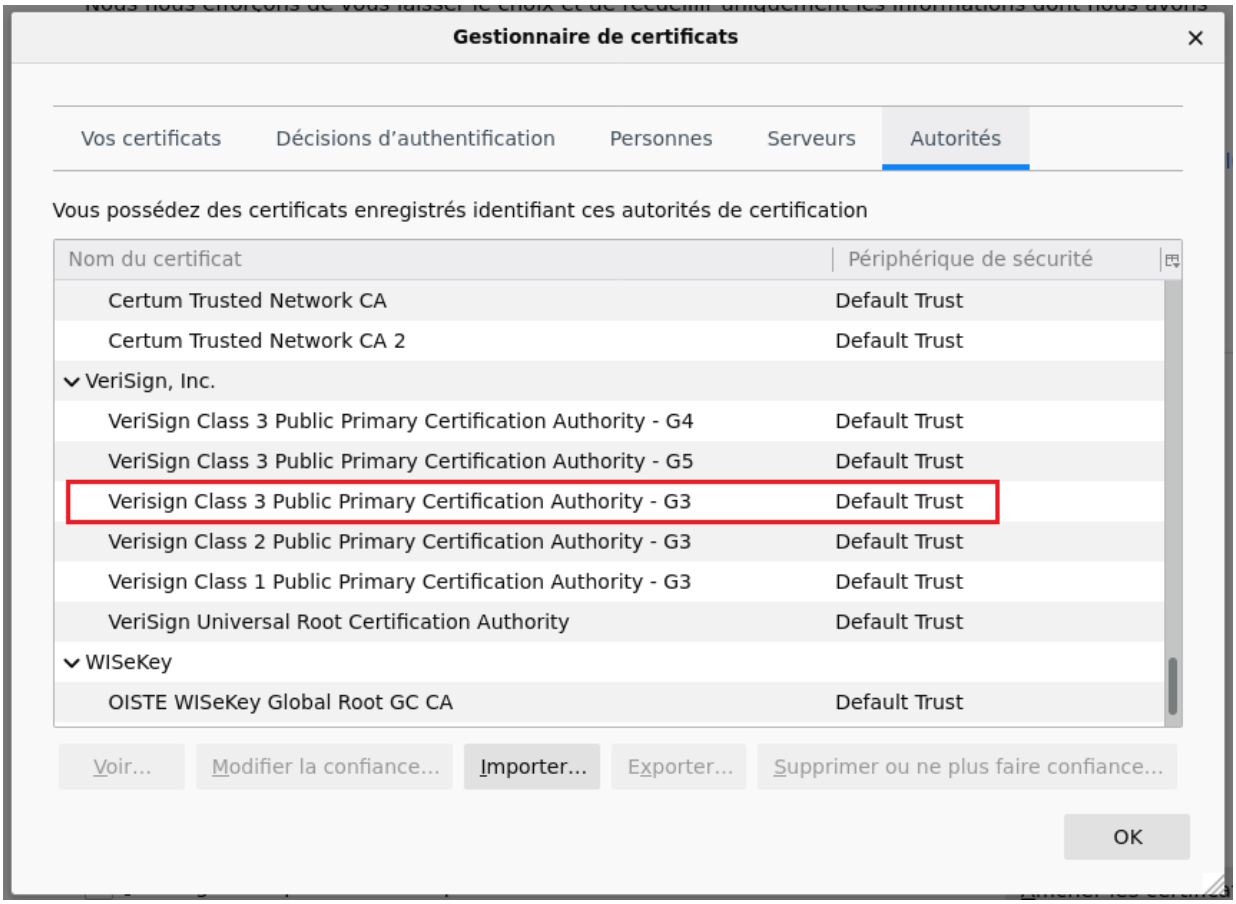
Exemple de lecture d'un fichier certificat sur disque

Le petit programme suivant va permettre d'ouvrir un fichier certificat dont le nom est passé en paramètre de la ligne de commande et d'en afficher les informations :

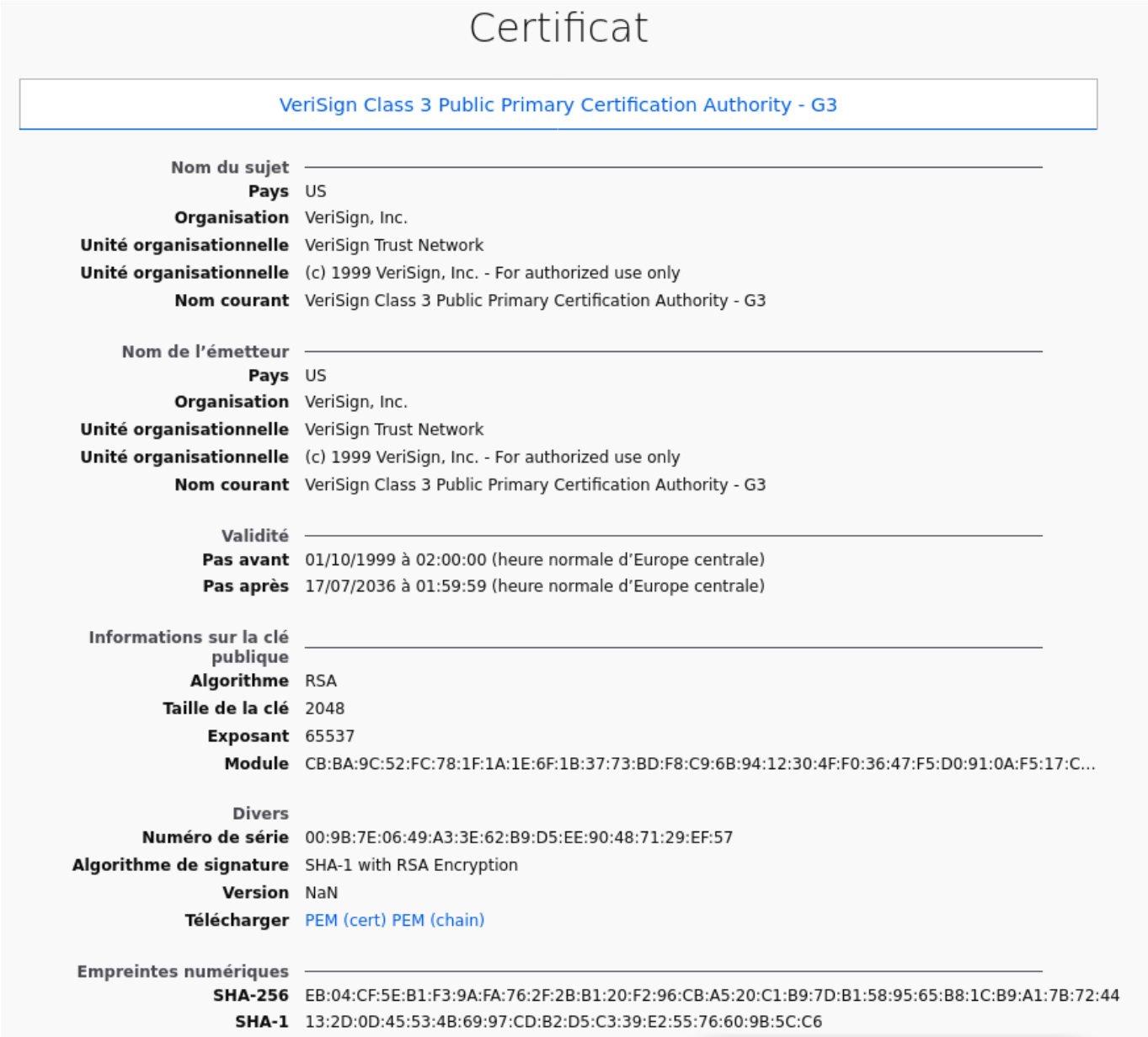
```
import java.io.*;
import java.security.*;
import java.security.cert.*;

public class TestCertificat
{
    public static void main(String args[]) throws …
    {
        InputStream inStream = null;
        instream = new FileInputStream(args[0]);
```

```
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        X509Certificate cert = (X509Certificate)cf.generateCertificate(inStream);

        System.out.println("Classe instanciée : " + cert.getClass().getName());
        System.out.println("Type de certificat : " + cert.getType());
        System.out.println("Nom du propriétaire du certificat : " +
cert.getSubjectDN().getName());

        PublicKey clePublique = cert.getPublicKey();
        System.out.println("... sa clé publique : " + clePublique.toString());
        System.out.println("... la classe instanciée par celle-ci : " +
clePublique.getClass().getName());
        System.out.println("Dates limites de validité : [" + cert.getNotBefore() + "
- " + cert.getNotAfter() + "]");
        System.out.println("Signataire du certificat : " +
cert.getIssuerDN().getName());
        System.out.println("Algo de signature : " + cert.getSigAlgName());
        System.out.println("Signature : " + cert.getSignature());
    }
}
```

Pour l'exemple d'exécution, il nous faut un certificat. Pour cela, nous nous procurons un certificat du CA « **Verisign** » dans le browser **Firefox** :



dont les renseignements détaillés sont



Certificat

VeriSign Class 3 Public Primary Certification Authority - G3

Nom du sujet	
Pays	US
Organisation	VeriSign, Inc.
Unité organisationnelle	VeriSign Trust Network
Unité organisationnelle	(c) 1999 VeriSign, Inc. - For authorized use only
Nom courant	VeriSign Class 3 Public Primary Certification Authority - G3
Nom de l'émetteur	
Pays	US
Organisation	VeriSign, Inc.
Unité organisationnelle	VeriSign Trust Network
Unité organisationnelle	(c) 1999 VeriSign, Inc. - For authorized use only
Nom courant	VeriSign Class 3 Public Primary Certification Authority - G3
Validité	
Pas avant	01/10/1999 à 02:00:00 (heure normale d'Europe centrale)
Pas après	17/07/2036 à 01:59:59 (heure normale d'Europe centrale)
Informations sur la clé publique	
Algorithme	RSA
Taille de la clé	2048
Exposant	65537
Module	CB:BA:9C:52:FC:78:1F:1A:1E:6F:1B:37:73:BD:F8:C9:6B:94:12:30:4F:F0:36:47:F5:D0:91:0A:F5:17:C...
Divers	
Numéro de série	00:9B:7E:06:49:A3:3E:62:B9:D5:EE:90:48:71:29:EF:57
Algorithme de signature	SHA-1 with RSA Encryption
Version	NaN
Télécharger	PEM (cert) PEM (chain)
Empreintes numériques	
SHA-256	EB:04:CF:5E:B1:F3:9A:FA:76:2F:2B:B1:20:F2:96:CB:A5:20:C1:B9:7D:B1:58:95:65:B8:1C:B9:A1:7B:72:44
SHA-1	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6

On remarque que le nom du sujet (le propriétaire de la **clé publique**) et le nom de l'émetteur (le CA) sont identiques. Cela veut dire que le certificat contient une clé publique de Verisign dont le certificat est signé par Verisign lui-même → on parle de **certificats autosignés**

Le téléchargement du certificat nous fournit le fichier **verisign-class-3-public-primary-certification-authority--g3.pem** que nous analysons avec notre programme :

```
# java TestCertificat verisign-class-3-public-primary-certification-authority--g3.pem
Classe instanciée : sun.security.x509.X509CertImpl
Type de certificat : X.509
Nom du propriétaire du certificat : CN=VeriSign Class 3 Public Primary Certification
Authority - G3, OU="(c) 1999 VeriSign, Inc. - For authorized use only", OU=VeriSign
Trust Network, O="VeriSign, Inc.", C=US
... sa clé publique : Sun RSA public key, 2048 bits
  modulus:
25718397044994404362896617983225066523605680143532218711497629578889668427069372903246969642705193718904440253681165922342854265566093074338881471099308420666768860527307833174615055705857927494993688023412080176891665806340198922575599798174795786894721928038442557145741856951721986037747496300227338936740935160614873112339916704017780522290710699461101873080668720181520041467631183634345286227350856664176524313780731440444145137601165359023778199735027017232749399655667940809534628879671818212244693110541386687290730654808469145618420223241822255502173639819741862996462324124465938488574888443972871289132951
  public exponent: 65537
... la classe instanciée par celle-ci : sun.security.rsa.RSAPublicKeyImpl
Dates limites de validité : [Fri Oct 01 02:00:00 CEST 1999 - Thu Jul 17 01:59:59 CEST
2036]
Signataire du certificat : CN=VeriSign Class 3 Public Primary Certification Authority
- G3, OU="(c) 1999 VeriSign, Inc. - For authorized use only", OU=VeriSign Trust
Network, O="VeriSign, Inc.", C=US
Algo de signature : SHA1withRSA
Signature : [B@55f96302
#
```

Voici un autre exemple d'exécution avec un certificat obtenu à partir d'une carte d'identité électronique belge :

```
# java TestCertificat jeanmarc_wagner_authentication.der
Classe instanciée : sun.security.x509.X509CertImpl
Type de certificat : X.509
Nom du propriétaire du certificat : SERIALNUMBER=74020722368, GIVENNAME=Jean-Marc
Christian, SURNAME=Wagner, CN=Jean-Marc Wagner (Authentication), C=BE
... sa clé publique : Sun RSA public key, 2048 bits
  modulus:
19995146287835309904425888244120705510147429119940212259509930435495930070433471265845906310990717572770766507831884036733011136523993788398145611322418795889218632829358282470698956242845307563306174103460670656203644795525218429470761325586750987412232924376072082456944705881353802042839737561054848517306576641358132361515620438033085724573704300424587317562570574271105971290253301588941078495093331082234080394851367204820298229917542060089158437329135902586509610845139840975504346952500910573963747744918637909367790348607071088970132321904266315844360789465595228803098721327793703560286429752418326384004397
  public exponent: 65537
... la classe instanciée par celle-ci : sun.security.rsa.RSAPublicKeyImpl
Dates limites de validité : [Sat Oct 17 00:10:45 CEST 2020 - Wed Oct 16 01:59:59 CEST
2030]
```

```
Signataire du certificat : SERIALNUMBER=202001, CN=Citizen CA, O=Certipost N.V./S.A., L=Brussels, C=BE
Algo de signature : SHA256withRSA
Signature : [B@3d4eac69
#
```

Ici il s'agit bet et bien d'un certificat signé par un CA appelé « Certipost ».

Les keystores

Principe et généralités

Il s'agit à présent, pour chaque intervenant d'une communication sécurisée, de disposer d'un système de stockage permettant de conserver :

- Ses **clés privées** (il peut en avoir plusieurs pour divers usages), ainsi que les **certificats associées** (contenant les clés publiques associées)
- Les **certificats** des personnes de confiance avec qui il veut communiquer

Java a mis en place un système de conteneurs pour cela, appelés **keystores**.

Un **keystore** est une sorte de dictionnaire qui contient, de manière cryptée, deux types d'entrées :

- **Key Entry** : une clé privée et une liste de certificats associés à la clé publique correspondante
- **Trusted Certificate Entry** : un certificat d'un intervenant considéré comme sûr

Chaque entrée de ce dictionnaire est identifiée à l'aide d'un **alias**.

Par exemple, on pourrait imaginer que l'intervenant JeanMarc dispose d'un keystore contenant les entrées suivantes :

Alias	Information	Usage
cleDocuments	Clé privée + clé publique + certificat(s)	Signer des documents
cleMails	Clé privée + clé publique + certificat(s)	Signer des emails
Virginie	Certificat	Authentifier
Claude	Certificat	Authentifier
Christophe	Certificat	Authentifier

Un **keystore** est en réalité un fichier propriétaire et la gestion interne d'un keystore est laissé aux soins des providers.

Il existe plusieurs formats de **keystore** :

- **JKS** : implémentation standard de Sun, avec un mot de passe distinct pour chaque clé privée
- **JCEKS** : amélioration de JKS → peut contenir des clés secrètes (symétriques)
- **PKCS12** : implémentation de Public-Key Cryptography Standards de RSA, un seul mot de passe global est nécessaire
- **BKS** : implémentation de Bouncy Castle, similaire à JCEKS
- **UBER** : amélioration du précédent, avec utilisation renforcée du mot de passe

Manipuler un keystore en ligne de commande

Pour cela, Java propose un utilitaire en ligne de commande appelé **keytool**. Celui-ci se trouve dans le répertoire bin du JDK. Cette commande présente plusieurs options :

- **-genkeypair** → permet de créer une entrée de type Key Entry
- **-keyalg …** → précise l'algorithme de génération de clé (RSA, DSA, …)
- **-alias …** → précise l'alias pour l'entrée en question
- **-storetype …** → précise le type de keystore
- **-keysize …** → précise la taille de la clé (en bits)
- **-dname …** → permet de définir un distinguished name (DN) pour cette entrée
- **-keystore …** → permet de préciser le nom du fichier keystore
- **-list** → affiche le contenu du keystore
- **-export** → permet d'exporter un certificat
- **-file …** → permet de préciser un nom de fichier
- **-printcert** → afficher le contenu d'un certifcat
- **-certreq** → permet de générer un fichier de demande de certification
- **-keypass …** → préciser le mot de passe associé à la clé privée
- **-storepass …** → préciser le mot de passe du keystore
- **-import** → importer un certificat

Si l'option **-keystore** n'est pas utilisée, un fichier keystore par défaut, appelée **.keystore**, est utilisé. Celui-ci se trouve automatiquement dans le répertoire $HOME de l'utilisateur. Il est donc préférable d'utiliser l'option -keystore.

Exemple de création d'une paire de clés privée/publique :

L'utilisateur Jean-Marc souhaite créer un keystore et une paire de clés privée/publiques :

```
# keytool -genkeypair -alias JeanMarc -keyalg RSA -keysize 2048 -dname "CN=Jean-Marc
Wagner,O=HEPL,C=B" -keystore KeystoreJeanMarc.jks
Entrez le mot de passe du fichier de clés :
Ressaisissez le nouveau mot de passe :
Entrez le mot de passe de la clé pour <JeanMarc>
      (appuyez sur Entrée s'il s'agit du mot de passe du fichier de clés) :
Ressaisissez le nouveau mot de passe :
# ls -l
...
-rw-rw-r--. 1 student student 2138 15 fév 10:45 KeystoreJeanMarc.jks
...
# keytool -keystore KeystoreJeanMarc.jks -list
Entrez le mot de passe du fichier de clés :

Type de fichier de clés : JKS
Fournisseur de fichier de clés : SUN

Votre fichier de clés d'accès contient 1 entrée

jeanmarc, 15-févr.-2022, PrivateKeyEntry,
Empreinte du certificat (SHA1) :
B1:EE:84:80:69:FF:DE:77:94:4B:B1:65:C6:00:A8:9F:8A:47:7F:AE
#
```

Dans cet exemple, une entrée de type **Key Entry** a été créée dans un keystore

- de type **JKS** (celui par défaut précisé dans le fichier java.security → implémentation par défaut de Sun) dont le nom est « KeystoreJeanMarc.jks »
- dont l'alias est **JeanMarc**
- pour un couple de clés de type **RSA** de taille 2048 bits
- Mot de passe Keystore : « **PassJeanMarc** »
- Mot de passe alias JeanMarc : « **PassCleJeanMarc** »

L'« empreinte du certificat (SHA1) » correspond à la signature du certificat.

Le certificat est signé… mais par qui ? Par la clé privée générée et stockée pour cet alias → il s'agit donc d'un **certificat autosigné** ! → tout se passe donc comme si JeanMarc avait signé son propre certificat…

Exemple d'exportation d'un certificat :

Le propriétaire d'une clé privée contenue dans un keystore peut exporter un certificat pour cette clé afin de la distribuer à d'autres intervenants. Dans l'exemple qui suit, Jean-Marc va exporter le certificat associé à sa clé privée d'alias JeanMarc.

```
# keytool -export -alias JeanMarc -file JeanMarc.cer -keystore KeystoreJeanMarc.jks
Entrez le mot de passe du fichier de clés :
Certificat stocké dans le fichier <JeanMarc.cer>
# ls -l
...
-rw-rw-r--. 1 student student  783 15 fév 11:14 JeanMarc.cer
-rw-rw-r--. 1 student student 2138 15 fév 10:45 KeystoreJeanMarc.jks
…
# keytool -printcert -file JeanMarc.cer
Propriétaire : CN=Jean-Marc Wagner, O=HEPL, C=B
Emetteur : CN=Jean-Marc Wagner, O=HEPL, C=B
Numéro de série : 25ee2cb2
Valide du : Tue Feb 15 10:44:35 CET 2022 au : Mon May 16 11:44:35 CEST 2022
Empreintes du certificat :
       MD5:  67:64:0D:6F:3E:E8:10:17:83:52:7F:32:24:E1:FD:1E
       SHA1 : B1:EE:84:80:69:FF:DE:77:94:4B:B1:65:C6:00:A8:9F:8A:47:7F:AE
       SHA256 :
94:0F:C8:B1:E3:7C:2D:48:84:1C:A4:A1:56:E3:90:05:76:81:FF:28:8C:5D:C5:B6:D1:82:13:0B:C
7:EE:13:6A
       Nom de l'algorithme de signature : SHA256withRSA
       Version : 3

Extensions :

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: C1 5D 78 03 14 1F 39 43   32 F5 15 25 53 A4 0B B7  .]x...9C2..%S...
0010: 6B EF 7F B6                                        k...
]
]

#
```

Le fichier certificat s'appelle donc **JeanMarc.cer** et contient la clé publique de Jean-Marc.

Exemple d'importation d'un certificat :

Supposons que l'utilisateur Christophe dispose d'un keystore appelé **KeystoreChristophe.jks** (mot de passe « **PassChristophe** ») qui contient déjà sa clé privée (ainsi que les certificats associés) sous l'alias « **Christophe** » (mot de passe « **PassCleChristophe** »). Dans l'exemple qui suit, Christophe importe le certificat de Jean-Marc dans son keystore :

```
# keytool -import -keystore KeystoreChristophe.jks -alias JeanMarc -file JeanMarc.cer
Entrez le mot de passe du fichier de clés :
Propriétaire : CN=Jean-Marc Wagner, O=HEPL, C=B
Emetteur : CN=Jean-Marc Wagner, O=HEPL, C=B
Numéro de série : 25ee2cb2
Valide du : Tue Feb 15 10:44:35 CET 2022 au : Mon May 16 11:44:35 CEST 2022
Empreintes du certificat :
       MD5:  67:64:0D:6F:3E:E8:10:17:83:52:7F:32:24:E1:FD:1E
       SHA1 : B1:EE:84:80:69:FF:DE:77:94:4B:B1:65:C6:00:A8:9F:8A:47:7F:AE
       SHA256 :
94:0F:C8:B1:E3:7C:2D:48:84:1C:A4:A1:56:E3:90:05:76:81:FF:28:8C:5D:C5:B6:D1:82:13:0B:C
7:EE:13:6A
       Nom de l'algorithme de signature : SHA256withRSA
       Version : 3

Extensions :

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: C1 5D 78 03 14 1F 39 43   32 F5 15 25 53 A4 0B B7  .]x...9C2..%S...
0010: 6B EF 7F B6                                        k...
]
]

Faire confiance à ce certificat ? [non] :  oui
Certificat ajouté au fichier de clés
# keytool -keystore KeystoreChristophe.jks -list
Entrez le mot de passe du fichier de clés :

Type de fichier de clés : JKS
Fournisseur de fichier de clés : SUN

Votre fichier de clés d'accès contient 2 entrées

jeanmarc, 16-févr.-2022, trustedCertEntry,
Empreinte du certificat (SHA1) :
B1:EE:84:80:69:FF:DE:77:94:4B:B1:65:C6:00:A8:9F:8A:47:7F:AE
christophe, 16-févr.-2022, PrivateKeyEntry,
Empreinte du certificat (SHA1) :
5D:1E:FB:39:5A:08:EB:41:0A:7E:AD:3B:1D:A6:3B:80:AC:9E:7B:45
#
```

Exemple d'obtention d'un certificat certifié par un CA :

Pour l'instant, les certificats manipulés sont auto-signés. Pour obtenir un certificat certifié par un CA reconnu, Jean-Marc doit générer une demande :

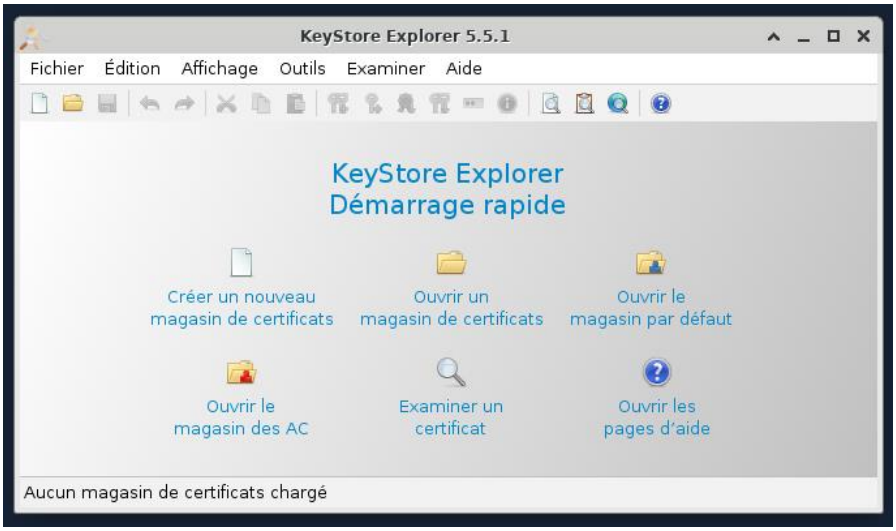
```
# keytool -certreq -alias JeanMarc -file JeanMarc.csr -keystore KeystoreJeanMarc.jks
Entrez le mot de passe du fichier de clés :
Entrez le mot de passe de la clé pour <JeanMarc>
# ls -l
…
-rw-rw-r--. 1 student student  783 15 fév 11:14 JeanMarc.cer
-rw-rw-r--. 1 student student 1023 17 fév 11:31 JeanMarc.csr
-rw-rw-r--. 1 student student 2138 15 fév 10:45 KeystoreJeanMarc.jks
…
#
```

Cette demande s'appelle un **CSR** (**C**ertificate **S**igning **R**equest). Ensuite,

1. Cette demande doit être transmise à un CA qui pourra alors vérifier, d'une manière ou d'une autre, le bien-fondé de la demande. Dans l'affirmative,
2. La CA fournit la certificat demandé → signé par lui
3. Jean-Marc (ou Christophe) peut alors importer ce certificat dans son keystore → si l'alias correspond à une entrée existante dans le keystore, le certificat initial, auto-signé, est remplacé par une chaîne de certificats au bout de laquelle se trouve le certificat du CA, le plus souvent auto-signé.

Manipuler un keystore à l'aide de « Keystore Explorer »

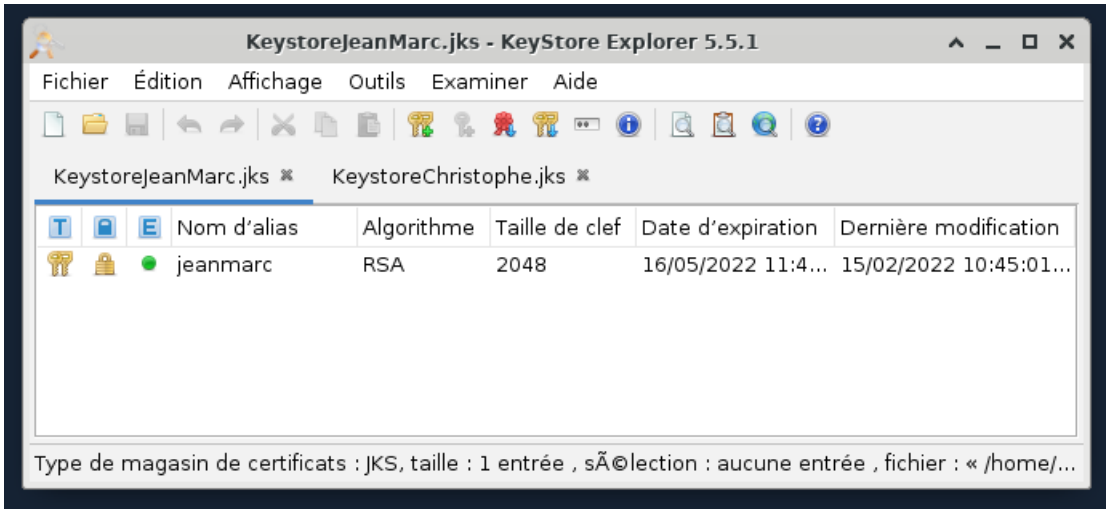
Il s'agit d'un logiciel graphique permettant de gérer les keystores. Au démarrage, il présente l'interface suivante :



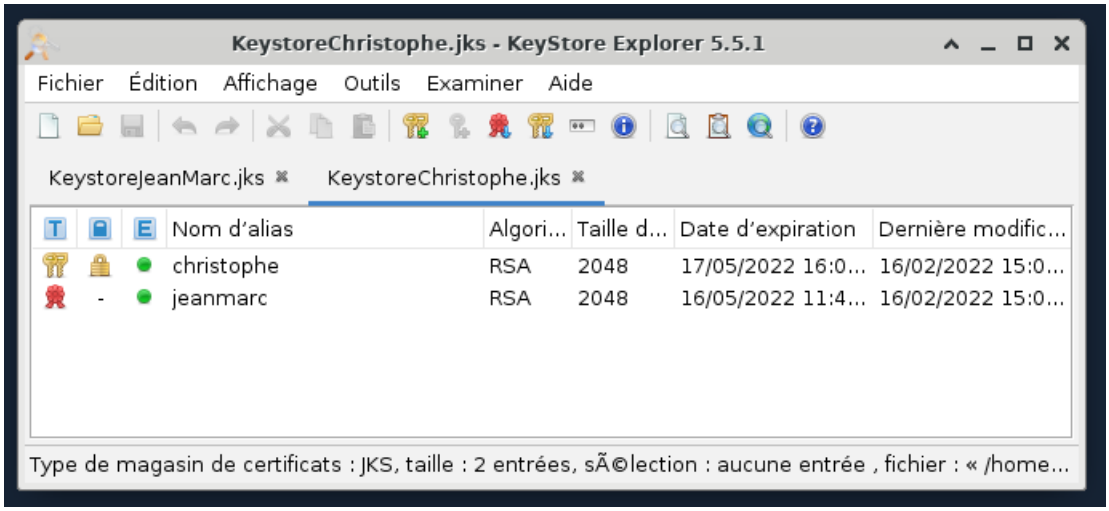
Il permet par exemple

- de créer un nouveau keystore
- d'ouvrir le keystore par défaut (fichier **.keystore** situé dans le répertoire de l'utilisateur)
- d'ouvrir un keystore dont on connaît le nom
- d'examiner un certificat
- créer des paires de clés privée/publique
- exporter des certificats
- …

Si, par exemple, on ouvre les keystores de Jean-Marc et de Christophe, cela donne

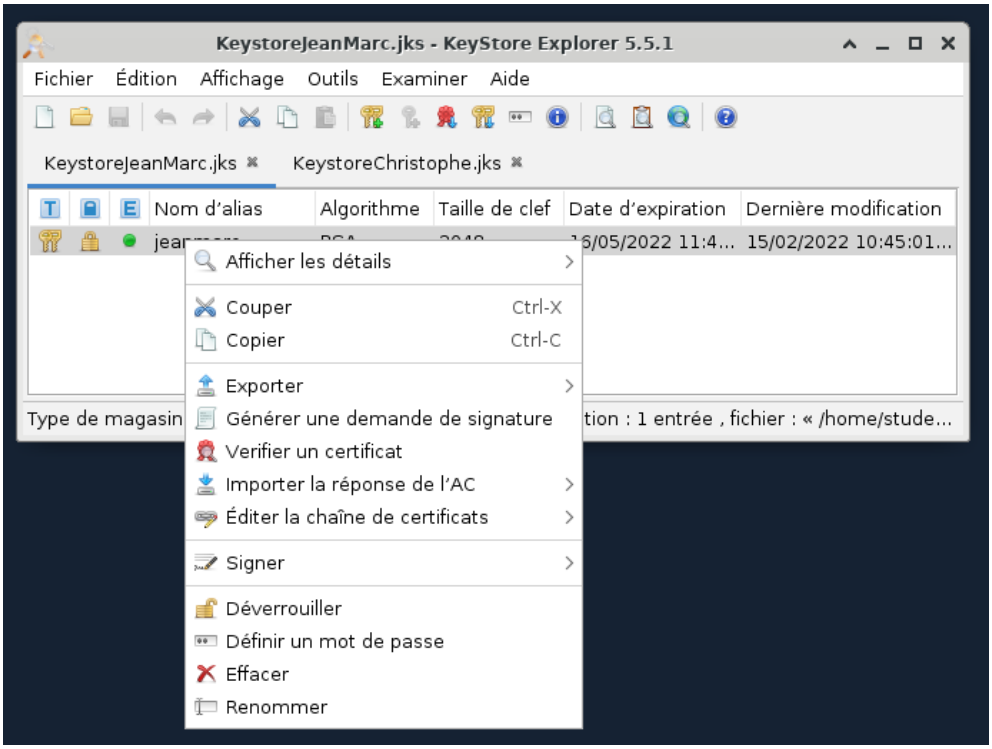


et

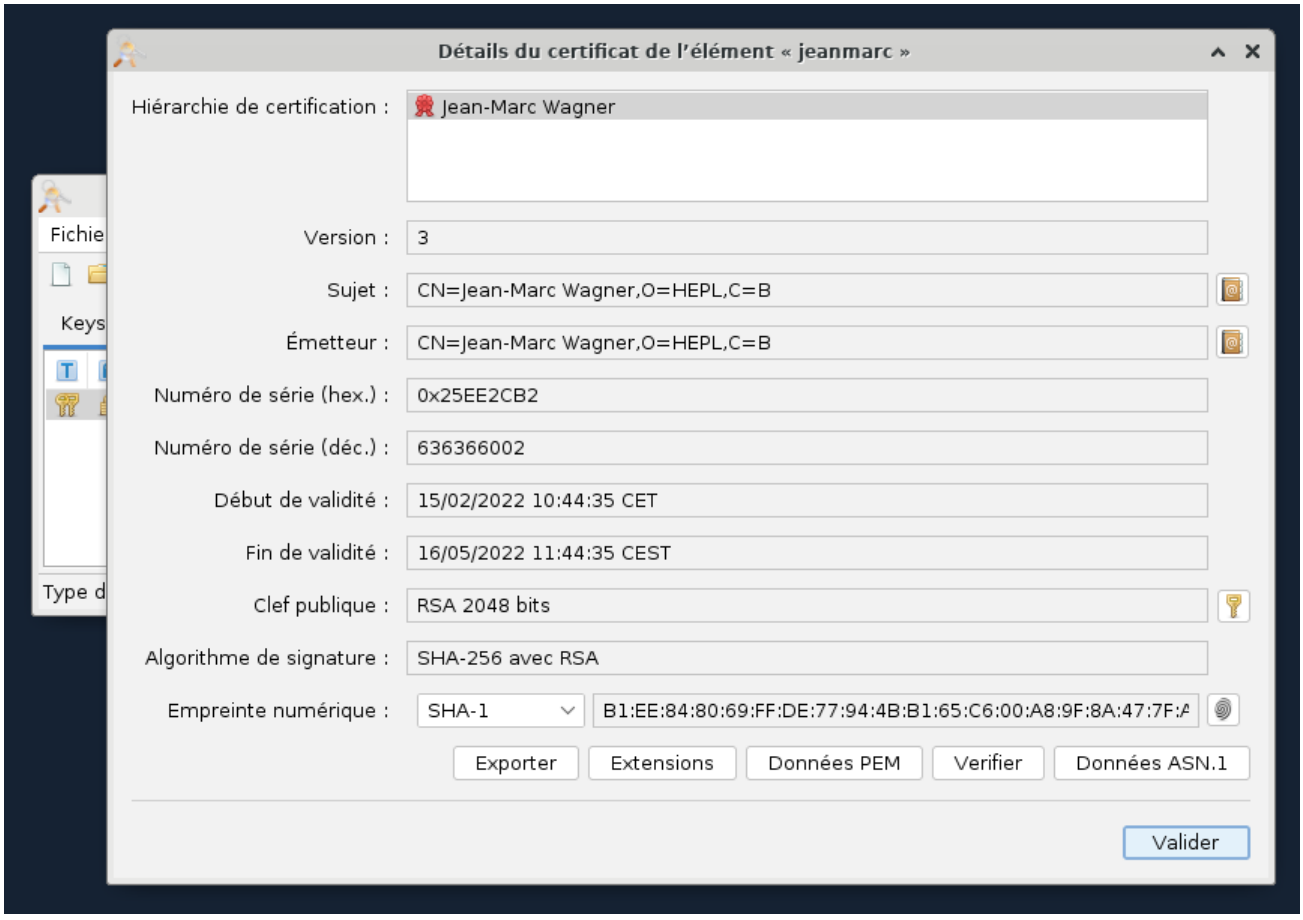


On y voit apparaître les différentes entrées, ainsi que leurs alias.

Un clic droit sur une entrée offre une multitude d'actions possibles :



comme par exemple obtenir tous les renseignements sur un certificat :



Manipuler un keystore en Java

Pour représenter un keystore en mémoire, Java propose la classe **KeyStore** du package java.security → l'implémentation est laissée à la discrétion des providers → il existe une implémentation par défaut fournie par Sun : le format **JKS**

On peut obtenir un objet keystore en utilisant la méthode de classe

- **public static KeyStore getInstance(String type)**
- **public static KeyStore getInstance(String type, String provider)**

où type représente le type de keystore souhaité (« jks », « pkcs12 », « jceks »,…) pour autant qu'un des providers en propose une implémentation.

On peut ajouter des entrées à un keystore en utilisant les méthodes de la classe **KeyStore** suivantes :

- **public final void setKeyEntry(String alias, byte[] key, Certificate[] chain)**
- **public final void setKeyEntry(String alias, Key key, char[] password, Certificate[] chain)**
- **public final void setCertificateEntry(String alias, Certificate cert)**

On peut également récupérer le contenu d'une entrée à l'aide des méthodes

- **public final Key getKey(String alias, char[] password)**
- **public final Certificate getCertificate(String alias)**

Pour charger le contenu d'un fichier keystore existant, on dispose de la méthode

- **public final void load(InputStream stream, char[] password)**

Où stream est un flux d'entrée pouvant clairement être associé à un fichier.

On peut enfin consulter le contenu d'un objet **KeyStore** à l'aide des méthodes

- **public final Enumeration<String> aliases()** → fournit la liste des alias
- **public final boolean isKeyEntry(String alias)** → permet de savoir si l'alias passé en paramètre est du type « KeyEntry »
- **public final boolean isCertificateEntry(String alias)** → permet de savoir si l'alias passé en paramètre est du type « CertificateEntry »

Exemple d'affichage du contenu d'un keystore

Le programme permettre d'afficher le contenu d'un keystore dont le nom du fichier sur disque et le mot de passe sont passés en paramètres en ligne de commande :

```
import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.util.*;

public class AfficheKeystore
{
    public static void main(String[] args) throws …
    {
        KeyStore ks = KeyStore.getInstance("JKS");
        ks.load(new FileInputStream(args[0]),args[1].toCharArray());

        Enumeration<String> en = ks.aliases();
        ArrayList<String> vec = new ArrayList<>();
        while (en.hasMoreElements()) vec.add(en.nextElement());

        for (String alias:vec)
        {
            if (ks.isKeyEntry(alias)) System.out.println("[keyEntry] --> " + alias);
            if (ks.isCertificateEntry(alias))
            {
                System.out.println("[trustedCertificateEntry] --> " + alias);

                X509Certificate certif = (X509Certificate)ks.getCertificate(alias);
                System.out.println("\tType de certificat : " + certif.getType());
                System.out.println("\tNom du propriétaire du certificat : " +
certif.getSubjectDN().getName());

                PublicKey clePublique = certif.getPublicKey();
                System.out.println("\tCle publique recuperee = " +
clePublique.toString());
            }
        }
    }
}
```

dont un exemple d'exécution fournit

```
# java Keystore.AfficheKeystore KeystoreChristophe.jks PassChristophe
[trustedCertificateEntry] --> jeanmarc
      Type de certificat : X.509
      Nom du propriétaire du certificat : CN=Jean-Marc Wagner, O=HEPL, C=B
      Cle publique recuperee = Sun RSA public key, 2048 bits
  modulus:
181694465829696039430848340714401108364750489966884576887137111471041733401663948106161144549664185880272686998621355128943331109414649989289473918196023388786643985091994078179451278551390992260792342701017527300646332382907265972965165075961308485227878
```

```
359401041124384930280324559265542386392948677193808162203975274173879984048194945240578363765477526234972320671386033098649380511694188123913920960842438323583069480802466335715714861479513593695735829755067384039235738439736055484958803262599545542941644919589390392145038992875205044514495142338703055458043768669885912276324408440328250599805761069539947801991
  public exponent: 65537
[keyEntry] --> christophe
#
```

Exemple d'utilisation d'un keystore dans un processus de signature

Nous reprenons ici l'exemple de signature RSA déjà utilisé plus haut. La seule différence est la technique de récupération des clés :

- **Jean-Marc** (le client) va récupérer sa **clé privée** dans son keystore
- **Christophe** (le serveur) va récupérer la **clé publique** de Jean-Marc dans son keystore

La classe **Requete** est totalement inchangée (fichier **Requete.java**) :

```
import java.io.*;
import java.security.*;

public class Requete implements Serializable
{
    private String nom;
    private int    age;
    private byte[] signature; // signature envoyée

    public Requete(String nom,int age,PrivateKey clePriveeClient) throws …
    {
        this.nom = nom;
        this.age = age;

        // Construction de la signature
        Signature s = Signature.getInstance("SHA1withRSA","BC");
        s.initSign(clePriveeClient);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeUTF(nom);
        dos.writeInt(age);
        s.update(baos.toByteArray());
        signature = s.sign();
    }

    public String getNom() { return nom; }
    public int getAge() { return age; }
```

```
    public boolean VerifySignature(PublicKey clePubliqueClient) throws …
    {
        // Construction de l'objet Signature
        Signature s = Signature.getInstance("SHA1withRSA","BC");
        s.initVerify(clePubliqueClient);
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeUTF(nom);
        dos.writeInt(age);
        s.update(baos.toByteArray());

        // Vérification de la signature reçue
        return s.verify(signature);
    }
}
```

Le programme client (fichier **ClientSignKeystore.java**) est :

```
import java.io.*;
import java.net.Socket;
import java.security.*;

public class ClientSignKeystore
{
    public static void main(String args[]) throws …
    {
        // Données à transmettre
        String nom = args[0];
        int age = Integer.parseInt(args[1]);
        System.out.println("Données à signer : Nom=" + nom + " Age=" + age);

        // Recuperation de la clé privée du client
        PrivateKey clePrivee = RecupereClePriveeClient();
        System.out.println("Récupération clé privée client...");

        // Construction de la requête
        Requete requete = new Requete(nom,age,clePrivee);

        // Connection sur le serveur
        Socket socket = new Socket("localhost",10000);
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        oos.writeObject(requete);
        oos.close();
        socket.close();
        System.out.println("Envoi de la requête");
    }

    public static PrivateKey RecupereClePriveeClient() throws …
    {
        // Récupération de la clé privée de Jean-Marc dans le keystore de Jean-Marc
        KeyStore ks = KeyStore.getInstance("JKS");
        ks.load(new
FileInputStream("KeystoreJeanMarc.jks"),"PassJeanMarc".toCharArray());
```

```
        PrivateKey cle = (PrivateKey)
ks.getKey("JeanMarc","PassCleJeanMarc".toCharArray());
        return cle;
    }
}
```

où seule la fonction **RecupereClePriveeClient()** a été mise à jour pour tenir compte du keystore de **Jean-Marc**.

Le programme serveur (fichier **ServeurSignKeystore.java**) est :

```
import java.io.*;
import java.net.*;
import java.security.*;

public class ServeurSignKeystore
{
    public static void main(String args[]) throws …
    {
        // Attente et réception de la requête
        System.out.println("Attente d'une requête...");
        ServerSocket socket = new ServerSocket(10000);
        Socket socketService = socket.accept();
        ObjectInputStream ois = new
ObjectInputStream(socketService.getInputStream());
        Requete requete = (Requete) ois.readObject();
        ois.close();
        socketService.close();
        socket.close();

        // Récupération de la clé publique du client
        PublicKey clePublique = RecupereClePubliqueClient();
        System.out.println("Récupération clé publique client...");

        // Récupération des données
        String nom = requete.getNom();
        int age = requete.getAge();

        System.out.println("Données à vérifier :");
        System.out.println("Nom = " + nom);
        System.out.println("Age = " + age);
        if (requete.VerifySignature(clePublique)) System.out.println("Signature
validée !");
        else System.out.println("Signature invalide...");
    }

    public static PublicKey RecupereClePubliqueClient() throws …
    {
        // Récupération de la clé publique de Jean-Marc dans le keystore de
Christophe
        KeyStore ks = KeyStore.getInstance("JKS");
```

```
        ks.load(new
FileInputStream("KeystoreChristophe.jks"),"PassChristophe".toCharArray());

        X509Certificate certif = (X509Certificate)ks.getCertificate("JeanMarc");
        PublicKey cle = certif.getPublicKey();
        return cle;
    }
}
```

où seule la fonction **RecupereClePubliqueClient()** a été modifiée pour tenir compte du keystore de **Christophe**.

Un exemple d'exécution du client est :

```
# java ClientSignKeystore Pierre 19
Données à signer : Nom=Pierre Age=19
Récupération clé privée client...
Envoi de la requête
#
```

tandis qu'au niveau du serveur (qui doit être lancé avant le client) :

```
# java ServeurSignKeystore
Attente d'une requête...
Récupération clé publique client...
Données à vérifier :
Nom = Pierre
Age = 19
Signature validée !
#
```

On pourrait encore imaginer que le client et le serveur disposent de keystore de formats différents…