Java Cryptography Architecture

O que é?

- É uma framework que facilita a incorporação de funcionalidades criptográficas em aplicações em Java.
- O que é um framework?
  - Uma framework orientada por objectos é um conjunto de classes que interagem entre si por forma a disponibilizar uma solução total ou parcial para um problema.
  - As frameworks permitem criar aplicações mais facilmente, sem exigir um grande conhecimento do domínio em que se aplicam.
  - Quando fornecem uma solução parcial para um problema (como neste caso) permitem reduzir o esforço de desenvolvimento, pois apenas é necessário acrescentar o código correspondente ás particularidades da aplicação a que vai ser aplicado.

Enquadramento

Código específico do problema a resolver
A desenvolver pelo programador

Aplicação

JCA

Código comum ao domínio de aplicação
Fornecido com o framework

Porque utilizar a JCA?

- Vantagens:
  - Que alternativas?
  - Independência das implementações específicas e dos algoritmos, compatibilidade entre as implementações e possibilidade de extensão.
  - Pertence à API base do Java e, como tal, permite escrever aplicações com técnicas criptográficas com todas as vantagens inerentes a esta linguagem (por exemplo, independência da plataforma).
  - Sendo um framework permite disponibilizar novas técnicas criptográficas com o mínimo de esforço.

Porque utilizar a JCA? (cont)

- Vantagens (cont.):
  - Algumas companhias com importância na área (e.g. a RSA Data Security, Inc.) já começam a desenvolver *providers* compatíveis com a JCA.
  - Existem outros frameworks escritos em Java com aplicação em domínios de interesse para a criptografia e.g. OCF.
- Desvantagens:
  - Pouco eficiente.
  - Inicialmente, as extensões com os mecanismos de cifra não eram exportáveis para fora dos USA+Canada.
  - Hoje em dia estas restrições já não são tão severas, mas a integração de implementações na JCA passou a ser restrita àquelas que apresentam uma certificação da Sun.

Objectivos da JCA

- Independência das implementações específicas de técnicas criptográficas.
- Compatibilidade entre as implementações.
- Independência dos algoritmos.
- Possibilidade de extensão.

Independência de implementações específicas

- A independência das implementações específicas implica que as aplicações possam usar um determinado algoritmo criptográfico sem se preocuparem com as especificidades das suas várias implementações.
- Na JCA esta independência consegue-se com o conceito de *provider*, que permite a co-existência de diversas implementações de subconjuntos da API num sistema, sendo isto transparente (até certo ponto) para a aplicação.
- Dado que a JCA normaliza a forma de nomear os diversos algoritmos criptográficos, quando uma aplicação requisita um determinado algoritmo ao framework é-lhe fornecida uma implementação de qualquer um dos providers instalados.

Independência de implementações específicas (cont)

- Desta forma, se a mesma aplicação for executada em plataformas com providers diferentes instalados, desde que estes implementem o mesmo algoritmo não existe qualquer problema.
- Esta independência também permite que os providers possam ser actualizados transparentemente.
- Por questões de segurança pode ser desejável não desenvolver aplicações independentes das implementações específicas. Ao requisitar um algoritmo podemos, opcionalmente, indicar qual o provider a que a implementação deve pertencer.

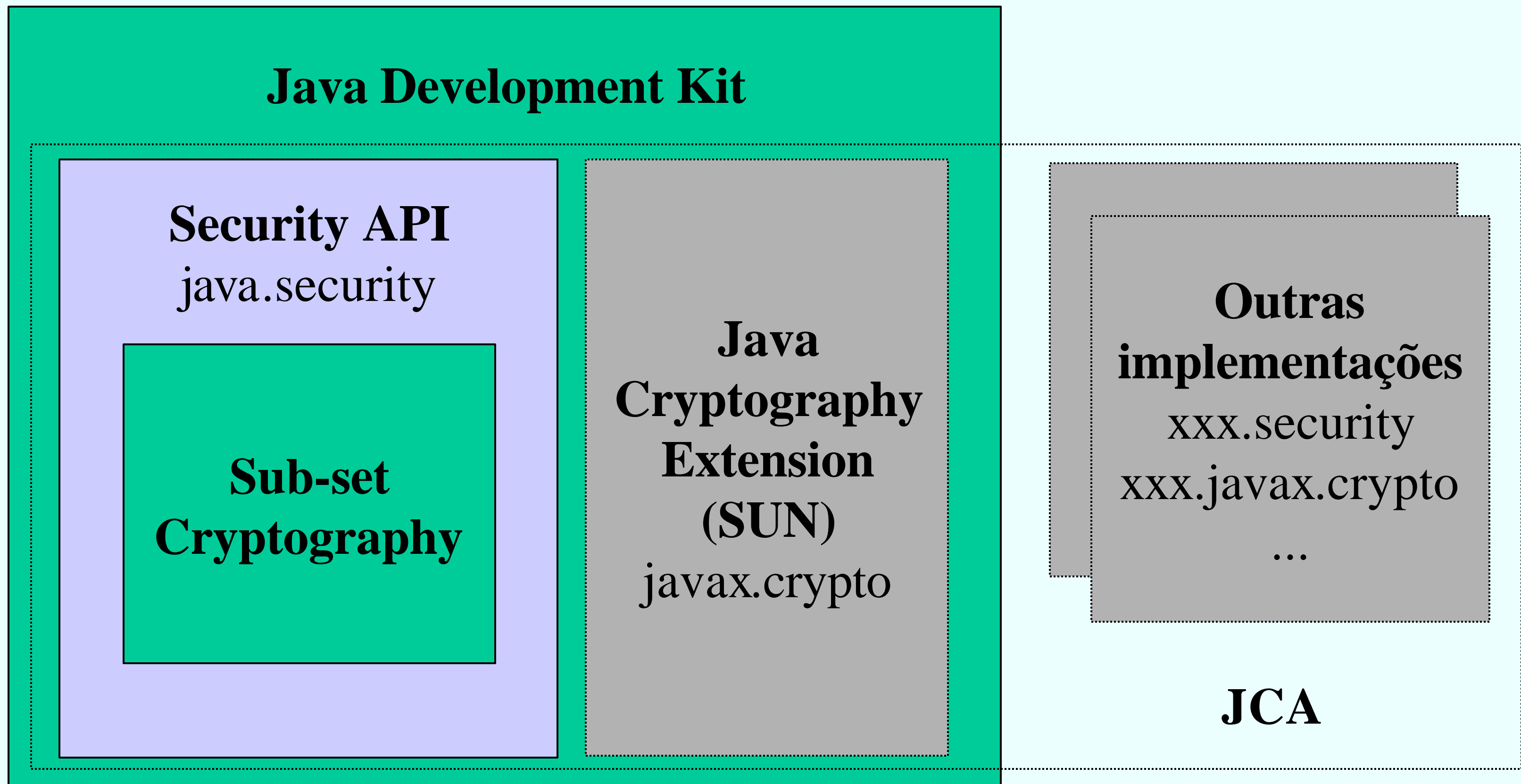
Independência dos algoritmos

- A independência dos algoritmos significa que os utilizadores podem utilizar as diversas técnicas criptográficas independentemente do algoritmo que as implementa.
- Esta independência é conseguida à custa de classes abstractas que especificam cada uma das técnicas criptográficas (e.g. mecanismos de hash, cifras ou assinaturas digitais).
- Estas classes são denominadas engine classes.
- As implementações concretas disponibilizadas nos providers devem respeitar a interface definida na engine class que especifica a técnica criptográfica do algoritmo implementado.

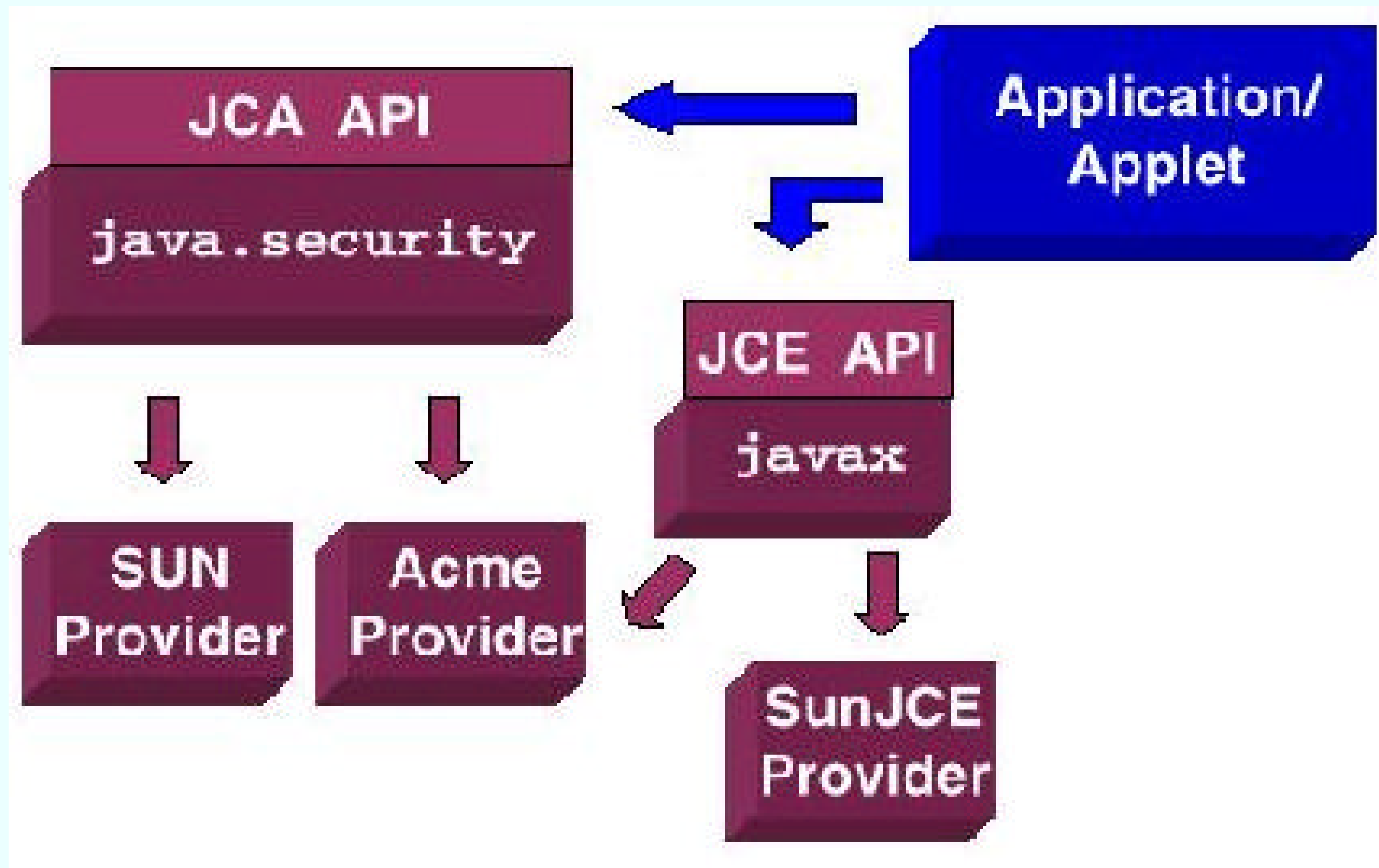
Compatibilidade entre implementações e possibilidade de extensão

- A compatibilidade entre as implementações permite que, por exemplo, para o mesmo algoritmo, as chaves geradas por um provider possam ser utilizadas por outro; ou que as assinaturas geradas por um provider possam ser verificadas por outro.
- Mais uma vez isto é conseguido à custa de classes que especificam os diversos componentes dos algoritmos criptográficos.
- A possibilidade de extensão significa que novos providers podem ser acrescentados ao framework. Estes providers devem disponibilizar implementações das engine classes já especificadas.

Componentes da JCA



Arquitectura da JCA



Providers Criptográficos

- A JCA introduz o conceito de Provider Criptográfico: um pacote (ou conjunto de pacotes) que fornecem uma implementação concreta de um subconjunto dos serviços definidos na API.
- No JDK 1.1 um provider podia conter uma implementação de um ou mais algoritmos de assinatura digital, algoritmos de hash e algoritmos de geração de chaves.
- No Java 2 aparecem cinco tipos adicionais de serviços: fábricas de chaves, criação e gestão de keystores, geração e gestão de parâmetros de algoritmos e fábricas de certificados.
- Aparece também a possibilidade de implementar algoritmos de geração de números pseudo-aleatórios.

O provider “SUN”

- O JDK inclui no seu package java.security a implementação de um provider minimal com o nome “SUN”.
- Este provider inclui implementações de:
  - Digital Signature Algorithm (DSA).
  - Algoritmos de hash MD5 e SHA-1.
  - Gerador de pares de chaves para DSA.
  - Geração e gestão de parâmetros para DSA.
  - Key factory capaz de instanciar objectos com chaves DSA.
  - Gerador de números pseudo-aleatórios "SHA1PRNG".
  - Certificate factory capaz de lidar com certificados X.509.
  - Keystore do tipo "JKS".

Outros providers

- Todos os providers implementam uma classe derivada da classe **Provider** especificada no package java.security.
- O construtor desta classe encarrega-se de registar na JCA todas as implementações suportadas pelo pacote correspondente.
- Isto permite à JCA reconhecer quais são os serviços e as implementações que estão disponíveis no sistema, e quais as livrarias a que pertencem, de modo a poder instanciar os objectos correspondentes, quando necessário.
- A instalação de um novo provider consiste em tornar visível esta classe à framework.

Instalação de um novo provider

- Antes de utilizar todas as potêncialidades da JCA é necessário adquirir ou desenvolver um provider criptográfico.
- Este é geralmente apresentado sob a forma de um ficheiro JAR (Java Archive), que contém todos os packages e que tem de estar na variável de ambiente CLASSPATH.
- Depois é necessário indicar à JCA a presença do novo provider, explicitando a sua classe principal. Isto pode ser feito de duas formas:
  - Estática
  - Dinâmica

A classe Security

- A classe Security efectua a gestão dos providers instalados e as propriedades de segurança do sistema. Esta classe tem apenas métodos estáticos, pelo que nunca é instanciada.
- Esta classe pode ser utilizada para obter uma lista dos providers instalados. O método **getProviders** retorna um array com todas as sub-classes Provider existentes no sistema, por ordem de preferência.
- O método **getProvider** permite obter um provider específico através do seu nome.
- A nível das propriedades de segurança, esta classe mantém uma lista destas propriedades, lista esta que está acessível através dos métodos **getProperty** e **setProperty**.

Instalação estática de um provider

- A instalação estática de um provider criptográfico faz com que ele seja carregado sempre que a JCA é activada, tornando-se visível através da classe Security.
- Este tipo de instalação implica a edição do ficheiro java.security, localizado na directoria /lib/security do JDK, ao qual deve ser adicionada uma linha com o seguinte formato:

  security.provider.n = classe.principal

- O valor de n indica a prioridade que vai ser dada ao novo provider. Quanto menor for n, maior é a nossa preferência pela utilização das implementações contidas neste provider.

Instalação dinâmica de um provider

- Sempre que uma aplicação necessita de um provider não referenciado no ficheiro java.security, pode instala-lo dinamicamente em run-time utilizando o método **Security.addProvider**:

```
classe.principal provider = new classe.principal();
prioridade=Security.addProvider(provider);
```

- O método **Security.insertProviderAt** pode ser utilizado para instalar um novo provider, atribuindo-lhe um determinado grau de preferencia.
- Para remover um provider usa-se o método **Security.removeProvider**.

Engine Classes

- Uma "engine class" define um serviço criptográfico de uma forma abstracta i.e. independente do algoritmo e da implementação e define uma API para esses serviços.
- Um serviço está associado a uma operação ou a um tipo particular e, consoante o caso, pode:
  - Efectuar operações criptográficas como a assinatura digital ou o hashing;
  - Gerar/gerir material criptográfico (chaves ou parâmetros) necessário para efectuar operações criptográficas;
  - Gerar/gerir objectos de dados (keystores ou certificados) que encapsulam chaves criptográficas de uma forma segura.

Engine Classes (cont)

- Associadas às Engine Classes existem "Service Provider Interfaces" (SPI): classes abstractas que definem métodos que implementações desses serviços têm de especificar.
- Uma instancia de uma Engine Class, i.e. a classe API, encapsula uma instancia da classe SPI correspondente escondendo a implementação específica da aplicação.
- O nome de uma classe SPI é o mesmo da Engine Class correspondente seguido de Spi e.g. Signature -> SignatureSpi.
- Na implementação de um provider em que se pretenda fornecer um determinado serviço é necessário definir uma sub-classe da classe SPI correspondente e implementar todos os métodos virtuais.

Engine Classes (cont)

- Ao contrário das Engine Classes, as classes Spi estão relacionadas com algoritmos específicos. Esta informação é essencial e está acessível através da API da Engine Class.
- Compete à classe principal do Provider registar as implementações que disponibiliza, de modo a que a engine class correspondente as consiga invocar.
- No que diz respeito às Engine Classes que efectuam a instanciação de objectos com material criptográfico, existem dois tipos:
  - Factories – Instanciam objectos com base em informação já existente num formato qualquer.
  - Generators – Criam novas unidades de informação criptográfica e.g. chaves.

Engine Classes da JCA

- MessageDigest – Utilizada para calcular o hash (message digest) dos dados que lhe são passados.
- Signature – Utilizada para assinar dados e verificar assinaturas.
- KeyPairGenerator – Utilizada para gerar pares de chaves publicas e privadas compatíveis com um algoritmo de cifra específico.
- KeyFactory – Utilizada para converter chaves criptográficas genéricas do tipo Key em representações transparentes das mesmas chaves, e vice-versa.
- CertificateFactory – Utilizada para lidar com certificados de chave publica e Certificate Revocation Lists (CRLs).

Engine Classes da JCA (cont)

- KeyStore – Utilizada para gerar e gerir uma keystore. Uma keystore é uma base de dados de chaves. As chaves privadas são associadas com uma cadeia de certificados que que autentica a chave pública associada. Uma keystore também contém certificados emitidos por *trusted entities*.
- AlgorithmParameters – Utilizada para gerir parâmetros de um determinado algoritmo (incluindo coding/decoding).
- AlgorithmParameterGenerator – Utilizada para gerar um conjunto de parâmetros compatíveis com um determinado algoritmo.
- SecureRandom – Utilizada para gerar números aleatórios ou pseudo-aleatórios.

Acesso às implementações

- Cada Engine Class tem um método chamado **getInstance** que permite obter instancias da classe, correspondentes a implementações específicas do serviço.
- Este método recebe como parâmetro o nome do algoritmo desejado, e procura dentro dos providers instalados uma implementação desse algoritmo.
- Se o algoritmo existente for disponibilizado por mais do que um provider, será retornada a implementação do provider com maior preferência.
- Caso não haja nenhuma implementação disponível é assinalada uma excepção que deve ser tratada pela aplicação.

Acesso às implementações (cont)

- Exemplo:

    Signature dsa = Signature.getInstance("DSA")

- Este comando procura instanciar uma implementação do DSA a partir de qualquer um dos providers instalados.
- Outro método das Engine Classes, também chamado getInstance, permite requisitar uma implementação de um algoritmo de um provider específico:

    Signature dsa = Signature.getInstance("DSA","SUN")

Message Digest

- A classe MessageDigest é a Engine Class dedicada às operações criptográficas de Message Digest.
- Um Message Digest é um algoritmo que, com base numa mensagem codificada como uma sequencia de bytes de comprimento arbitrário, gera um número de comprimento fixo chamado hash ou digest.
- Um digest tem duas propriedades:
  - Deve ser intratável obter duas mensagens com o mesmo hash.
  - O hash não revela nada sobre a mensagem que o originou.
- Os hash são utilizados para gerar identificadores de dados fiáveis e únicos: impressões digitais de mensagens.

Message Digest (cont)

- Os algoritmos de hash mais comuns são:
  – MD5 – Produz um valor de 128 bits. Em primeiro lugar os dados são processados e expandidos de forma a obter-se um comprimento múltiplo de 512 bits. O texto resultante é dividido em blocos de 512 bits que são processados de forma cíclica. O hash final é computado com base em cálculos parciais não lineares efectuados em cada parcela.O seu nome na JCA é MD5.
  – SHA1 – Semelhante ao MD5 mas produz um hash de 160 bits, conferindo maior segurança. É aconselhado para o algoritmo DSA. O seu nome na JCA é SHA.

Message Digest (cont)

- Quando se obtém uma instancia de um objecto MessageDigest, a sua utilização baseia-se em três métodos:
  - update – usado para transferir para o mecanismo a mensagem que originará o hash. Podem ser feitas várias invocações do método, sendo a mensagem final constituída pela concatenação de todos os blocos.
  - digest – usado para calcular o hash. O resultado é retornado como um array de bytes. Quando o calculo termina o mecanismo está reinicializado.
  - reset – reinicializa o mecanismo.

SecureRandom

- Esta é a Engine Class dedicada ás operações de geração de números (pseudo) aleatórios que são essenciais em criptografia e.g. nos algoritmos de cifra não determinísticos.
- A utilização desta classe consiste simplesmente em obter uma instância e utilizar o método **nextBytes** para obter um array de bytes aleatórios.
- Por defeito, esta classe está preparada para tentar inicializar-se de forma totalmente aleatória. No entanto disponibiliza também um método para se efectuar a inicialização de sementes: **setSeed**.

SecureRandom (cont)

- No caso de não ser fornecida uma semente, a inicialização é feita por um objecto interno de geração de sementes aleatórias com base em medições de tempo nos threads do sistema.
- Esta classe dispõe também de um método para gerar sementes e.g. para inicializar outros geradores de números aleatórios: **generateSeed**. Neste caso é utilizado o objecto interno mencionado acima.
- O provider "SUN" fornece uma implementação de SecureRandom que gera sequencias de bytes aleatórias com base no algoritmo de hash SHA-1.
- O nome desse algoritmo é SHA1PRNG.

Chaves na JCA

- As chaves obtêm-se a partir de *key generators*, certificados, *key factories* baseadas numa determinada especificação, ou de uma base de dados KeyStore que pode ser utilizada para gerir um conjunto de chaves.
- A interface **Key** está no topo da hierarquia de classes que lidam com chaves de uma forma “opaca”. Define a funcionalidade partilhada por todos os objectos deste tipo.
- Numa representação “opaca” de uma chave temos acesso directo ao material que constitui a chave.
- Isto contrasta com as representações transparentes em que, através de métodos específicos, as diversas partes da chave nos são disponibilizadas numa API de nível superior.

Chaves na JCA (cont)

A interface Key define métodos que nos permitem aceder as propriedades básicas de qualquer chave:

– **getAlgorithm** – Retorna o algoritmo, ou família de algoritmos, no qual a chave pode ser utilizado e.g. DSA, cifra, etc.

– **getFormat**, **setFormat –** Quando uma chave é necessária fora da aplicação, ela tem de ser codificada numa forma reconhecível por terceiros. Formatos standard como o X.509 e o PKCS#8 podem ser utilizados. Estes métodos permitem configurar o objecto Key para utilizar um desses formatos.

– **getEncoded –** Este método permite obter o resultado dessa codificação.

Chaves na JCA (cont)

- As interfaces **SecretKey**, **PublicKey** e **PrivateKey** são extensões sem métodos adicionais à interface Key.
- Estas extensões são utilizadas para contextualizar a utilização das chaves.
- Por exemplo PublicKey e PrivateKey fazem sentido em algoritmos de assinatura digital e cifra assimétrica.
- SecretKey faz sentido em cifras simétricas.
- A classe KeyPair é simplesmente um encapsulamento de um par de chaves PublicKey e PrivateKey.

Chaves na JCA (cont)

- As representações transparentes de chaves chamam-se especificações. Neste tipo de representação, a especificidade de uma chave pode ser de vários tipos:
  - Se a chave está armazenada num dispositivo de hardware, a especificação poderá conter informação que identifique a chave nesse ambiente.
  - Para uma chave associada a um determinado tipo de algoritmo, a especificação inclui métodos do tipo getxxx que permitem aceder aos seus componentes.
  - Uma chave pode estar especificada num formato independente dos algoritmos e.g. o PKCS#8. Neste caso, a especificação inclui métodos relacionados com esse formato particular.

Chaves na JCA (cont)

- As específicações suportadas pela JCA estão definidas no pacote java.security.spec.
- A interface **KeySpec** não contém métodos nem constantes.
- O seu único objectivo é agrupar e proteger todas as especificações de chaves, que têm forçosamente de a implementar.
- As definições básicas da JCA incluem especificações para os seguintes tipos de chaves:
  - Chaves privadas e públicas para DSA e RSA.
  - Chaves codificadas em X.501 e PKCS#8.

KeyPairGenerator e KeyGenerator

- A Engine Class **KeyPairGenerator** permite gerar pares de chaves para algoritmos assimétricos.
- Quando se requisita uma instancia desta classe, o nome do mecanismo deve ser o do algoritmo a que se destinam as chaves.
- Estes objectos podem ser inicializados de duas formas, utilizando duas variantes do método **initialize**:
  – Específicamente para o algoritmo, indicando um conjunto completo de parâmetros numa instancia de uma subclasse de AlgorithmParameterSpec.
  – Independentemente do algoritmo, indicando uma fonte de números aleatórios (opcional) e o tamanho (força) da chave. Neste caso, os restantes parâmetros tomam valores por defeito, ou gerados internamente pela implementação.

KeyPairGenerator e KeyGenerator

- O método **genKeyPair** pode ser invocado para obter o par de chaves encapsulado num objecto do tipo **KeyPair**.
- A classe **KeyGenerator** é semelhante à classe anterior, sendo no entanto utilizada para gerar chaves para algoritmos simétricos.
- Como os algoritmos simétricos pertenciam ao conjunto de tecnologias restritas pela legislação dos EUA/Canadá, esta classe está definida no pacote javax.crypto.
- As chaves geradas pelo método **generateKey** desta classe são retornadas como objectos do tipo **SecretKey**.
- A inicialização da classe é feita de forma semelhante à KeyPairGenerator, utilizando o método **init**.

KeyFactory

- Esta Engine Class permite converter entre representações de chaves opacas e especificações particulares.
- Estas conversões são bidireccionais i.e. é possível obter um objecto do tipo Key a partir de uma representação transparente e vice-versa.
- Uma mesma representação opaca pode corresponder a diversas especificações e.g. uma chave pública pode ser vista da perspectiva do algoritmo que a utiliza (por exemplo o DSA) ou da perspectiva do sistema de codificação que utiliza (por exemplo o X.509).
- Os objectos KeyFactory fazem efectivamente um parsing das chaves para obter as suas componentes num determinado contexto.

KeyFactory (cont)

- A obtenção de uma instancia da classe faz-se da mesma forma, através do método **getInstance**, indicando o algoritmo pretendido para a especificação e.g. DSA.
- Os métodos **generatePublic** e **generatePrivate** permitem obter objectos do tipo PublicKey e PrivateKey a partir de especificações transparentes.

```
PrivateKey privKey = keyFactory.generatePrivate(dsaPrivKeySpec);
PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
```

- O método **getKeySpec** permite obter uma representação transparente específica a partir de uma representação opaca.

```
DSAPublicKeySpec dsaPubKeySpec = (DSAPublicKeySpec)
    keyFactory.getKeySpec (pubKey, DSAPublicKeySpec.class)
```

Parametrização de Algoritmos

- Como para as chaves, existem dois tipos de encapsulamento para parâmetros de algoritmos: opacas e transparentes.
- A classe **AlgorithmParameters** fornece uma representação opaca em que não há acesso directo aos diversos parâmetros. É apenas possível obter o nome do algoritmo a que correspondem e extrair uma codificação particular.
- Um outro conjunto de classes, que oferece representações transparentes, com acesso directo a cada um dos parâmetros através de métodos **getxxx** específicos.
- A este tipo de classes chama-se também especificações.
- A interface **AlgorithmParameterSpec**, que não contém métodos, agrupa todas as classes deste tipo.

AlgorithmParameters

- Na obtenção de uma instancia desta classe deve indicar-se o algoritmo a que se destinam os parâmetros em questão.
- O método **init** permite inicializar o objecto a partir de um array de bytes num formato específico. No caso de um formato não ser indicado, o ASN.1 é utilizado.
- O método **getEncoded** permite obter os parâmetros numa representação específica ou, por defeito, utilizando o ASN.1.
- O método **getParameterSpec** permite obter uma representação transparente dos parâmetros e.g. **DSAParameterSpec**.

AlgorithmParameterGenerator

- Esta Engine Class pode ser utilizada para gerar um novo conjunto de parâmetros para um determinado algoritmo que é especificado em **getInstance**.
- Tal como nas classes de geração de chaves, a inicialização de uma instancia pode ser explicita ou independente do algoritmo.
- No ultimo caso, assume-se que todos os algoritmos têm em comum os conceitos de um gerador de números aleatórios e de tamanho/força.
- No caso de explicitarem os parâmetros de inicialização, é utilizado um objecto que implemente a interface **AlgorithmParameterSpec**.
- A geração é feita com o método **generateParameters**.

Certificados

- Na JCA, os certificados são suportados através de uma classe abstracta chamada **Certificate**.
- Esta classe agrupa a funcionalidade comum a todos os tipos de certificados:
  - Todos os tipos (**getType**) de certificados incluem uma chave pública (**getPublicKey**), validada por uma qualquer entidade (uma pessoa, uma autoridade de certificação, etc.).
  - Todos os tipos de certificados permitem realizar operações como a verificação da sua validade utilizando a chave pública da entidade correspondente (**verify**), ou a sua codificação num determinado formato (**getEncoded**, **toString**, **hashCode**).
- Note-se que não há métodos para gerar certificados. Na JCA assume-se que os certificados são gerados por entidades externas.

Certificados (cont)

- Classes especializadas para um determinado tipo de certificado devem ser sub-classes de **Certificate**.
- Por exemplo, a classe X509Certificate que define um certificado X.509, disponibiliza métodos para aceder aos atributos definidos na versão 3 destes certificados.
- Alguns destes métodos são os seguintes:
  – **getSerialNumber** - número de série do certificado.
  – **getIssuerDN** - *distinguished name* da entidade emissora.
  – **getSubjectDN** - *distinguished name* associado à chave pública.
  – **getNotBefore** e **getNotAfter** - Período de validade.

Certificate Revokation Lists

- Na JCA, as CRLs são suportadas através de uma classe abstracta chamada **CRL**.
- Esta classe agrupa a funcionalidade comum a todos os tipos de CRLs:
  - Todos os tipos (**getType**) de CRLs incluem a funcionalidade de, dado um determinado certificado, na forma de um objecto do tipo Certificate, retornar um valor boleano indicador do seu estado de revogação, ou não (**isRevoked**).
  - Todos os tipos de CRLs permitem obter a sua representação num objecto do tipo String (**toString**).
- Classes especializadas para um determinado tipo de CRL devem ser sub-classes de **CRL**.

CertificateFactory

- Esta Engine Class permite gerar objectos que encapsulam certificados e Certificate Revokation Lists (CRLs).
- O tipo de certificados a gerar é indicado quando se obtém a instancia da fábrica e.g. X.509.
- Esta classe gera certificados a partir de uma InputStream.
- O método **generateCertificate** devolve um certificado.
- O método **generateCertificates** devolve uma colecção de certificados.
- Da mesma forma, os métodos **generateCRL** e **generateCRLs** devolvem uma CRL ou uma colecção de CRLs.

Gestão de chaves e certificados

- Uma base de dados chamada Key Store pode ser utilizada para armazenar chaves e certificados, utilizando uma interface de alto nível fornecida pela Engine Class KeyStore.
- Podem existir diversas implementações deste tipo de objecto e.g. as chaves podem ser guardadas num ficheiro cifrado ou num smart card.
- O Provider "SUN" inclui uma implementação de KeyStore chamada "JKS".
- O JDK traz diversas ferramentas de linha de comando que são baseadas na funcionalidade desta classe.
- Este tipo de key store protege as chaves privadas com uma password. Globalmente, a key store é protegida da mesma forma.

Gestão de chaves e certificados (cont)

- A classe KeyStore permite armazenar dois tipos de entrada:
  - Chaves (Keys) – Estas entradas podem ser do tipo SecretKey, ou então uma PrivateKey acompanhada da PublicKey correspondente na forma de uma cadeia de certificados atestando a sua validade.
  - Certificados (Trusted Certificates) – Estas entradas correspondem a certificados contendo chaves públicas pertencentes a terceiros. A designação "trusted" vem do facto de o utilizador confiar que as chaves públicas em questão pertencem ao '*subject*' do certificado.
- Cada entrada é identificada por um alias, um string que a identifica de uma forma familiar.
- No caso das chaves privadas, cada um desses aliases corresponde efectivamente a uma maneira diferente de o utilizador se identificar.

Gestão de chaves e certificados (cont)

- O método **load** permite carregar uma Key Store para a memória, com base num InputStream e numa password que protege a base de dados.
- No caso de o InputStream ser **null**, é criada uma nova Key Store.
- O método **aliases** retorna uma lista dos aliases de todos as entradas na Key Store.
- Os métodos **isKeyEntry** e **isCertificateEntry** permitem testar uma entrada quanto ao seu tipo, com base no seu alias.
- O método **deleteEntry** permite eliminar uma das entradas com base no seu alias.

Gestão de chaves e certificados (cont)

- O método **setCertificateEntry** permite adicionar (ou actualizar) o conteúdo de um objecto do tipo **Certificate** à Key Store, com um determinado alias.
- O método **setKeyEntry** permite fazer uma operação equivalente com base num objecto do tipo **Key** ou num array de bytes com um determinado formato.
- Por exemplo, no provider "SUN", este array de bytes tem de respeitar o standard PKCS#8.
- Vocacionado para chaves privadas, este método exige uma password e um array com a lista de certificados que valida a chave pública correspondente.
- No caso de se pretender armazenar uma chave secreta, a cadeia de certificados deve ser **null**.

Gestão de chaves e certificados (cont)

- Os métodos **getKey**, **getCertificate** e **getCertificateChain** permitem extrair informação da Key Store com base num alias. Apenas no primeiro caso é necessário indicar uma password, uma vez que as chaves privadas são as únicas a ser protegidas.
- Note-se que se pode pedir um certificado com base num alias de uma chave privada. Neste caso é retornado o primeiro certificado da Certificate Chain correspondente.
- O método **getCertificateAlias** permite procurar um determinado certificado (um objecto do tipo **Certificate**) na Key Store, retornando o seu alias.
- O método **store** permite armazenar a Key Store utilizando um objecto do tipo OutputStream.
- A password passada a este método é utilizada para gerar um checksum do ficheiro resultante, para assegurar a sua protecção.

Signature

- A classe **Signature** é a Engine Class dedicada à operação criptográfica de assinatura digital e verificação de assinaturas.
- Numa assinatura digital, uma chave privada e uma mensagem com comprimento arbitrário são utilizados para gerar uma *assinatura* que atesta a autenticidade e permite verificar a integridade dos dados.
- Uma assinatura á uma sequência relativamente curta de bytes com as seguintes propriedades:
  – Dada a chave publica correspondente à chave privada utilizada, deve ser possível verificar a autenticidade e a integridade da mensagem.
  – A assinatura e a chave publica não revelam nada sobre a chave privada.

Signature (cont)

- Como a classe Signature pode ser utilizada tanto para gerar como para verificar assinaturas, o seu funcionamento baseia-se numa variável de estado que indica a operação que se pretende realizar.
- O estado actual de uma instancia determina que operações se podem realizar.
- A variável de estado pode tomar 3 valores: UNINITIALIZED, SIGN ou VERIFY. UNINITIALIZED é o estado inicial em que nenhuma operação pode ser realizada.
- A escolha do estado de um objecto deste tipo é feito utilizando métodos de inicialização.

Signature (cont)

- Para inicializar uma implementação no modo SIGN utiliza-se o método **initSign** que recebe como parâmetro uma chave privada **PrivateKey**.
- Para inicializar uma implementação no modo VERIFY utiliza-se o método **initVerify** que recebe como parâmetro uma chave pública **PublicKey**.
- O método **update** permite passar para dentro do objecto os dados que se pretende assinar ou verificar. Esta transferência pode ser feita de uma só vez, ou por blocos, invocando esta função sucessivas vezes.

Signature (cont)

- Para efectuar uma assinatura utiliza-se o método **sign** que não recebe parâmetros e retorna um array de bytes contendo a assinatura solicitada.
- Essa assinatura está codificada de acordo com o convencionado para cada algoritmo específico e.g. para o DSA a assinatura é uma codificação de uma estrutura ASN.1 de dois inteiros **r** e **s**.
- Para verificar uma assinatura utiliza-se o método **verify** que recebe como parâmetro um array de bytes contendo a assinatura a verificar (no mesmo formato retornado por **sign**) e retorna um valor booleano com o resultado da verificação.

Signature (cont)

- Os algoritmos de assinatura digital mais comuns são:
  - DSA com SHA-1 (SHA1withDSA) – definido no standard FIPS PUB 186.
  - RSA com MD2 (MD2withRSA) – definido no standard PKCS#1.
  - RSA com MD5 (MD5withRSA) – também definido no standard PKCS#1.
  - RSA com SHA-1 (SHA1withRSA) – definido no standard OSI Interoperability Workshop, utilizando as convenções de padding descritas no PKCS#1.

Java Cryptography Extension

- Este pacote (**javax.crypto**) estende a funcionalidade do pacote **java.security**.
- Foi criado separadamente devido às restrições de exportação de tecnologias criptográficas dos EUA/Canadá. Todas as tecnologias abrangidas por esta restrição são implementadas apenas neste pacote.
- Originalmente, a impossibilidade de obter a implementação JCE da Sun era colmatada com a flexibilidade de integrar outras implementações na JCA.
- Hoje em dia já é possível a obter a implementação da Sun, mas a possibilidade de utilizar outras implementações (possivelmente melhores) mantém-se.

Tecnologias da JCE

- As tecnologias que são implementadas na JCE são as seguintes:
  - Algoritmos de cifra simétrica como o RC4.
  - Algoritmos de cifra simétrica por blocos como o DES, o RC2 e o IDEA.
  - Algoritmos de cifra assimétrica como o RSA.
  - Algoritmos de cifra baseados em passwords (password-based encryption - PBE).
  - Message Authentication Codes (MAC).
  - Key Agreement (acordo de chaves).

Implementações da JCE

- Sun
  - http://java.sun.com/security
  - Provider: com.sun.crypto.provider.SunJCE
- IAIK
  - http://jcewww.iaik.at/
  - Provider: iaik.security.provider.IAIK
- ABA
  - http://www-dse.doc.ic.ac.uk/Courses/netsec/
  - Provider: au.net.aba.crypto.provider.ABAProvider
- Outras
  - Cryptix, Entrust, Forge, RSA, etc.

Conceitos da JCE

- Cifrar e Decifrar
  - Cifrar é o processo de, com base numa chave, transformar um conjunto de dados contendo informação (*cleartext*), num conjunto de dados sem significado para uma terceira parte que não conheça a chave *(ciphertext)*.
  - Decifragem é o processo inverso.
- Cifra baseada em password (Password-Based Encryption)
  - Este tipo de algoritmos de cifra geram a chave com base numa password.
  - Para tornar este processo mais seguro, a maior parte dos algoritmos deste tipo introduzem factores aleatórios no processo de geração da chave (*salt)*.

Conceitos da JCE (cont)

- Message Authentication Code
  - Permite verificar a integridade de informação transmitida ou armazenada num meio não confiável.
  - Tipicamente os MACs são utilizados entre duas partes que partilham uma Secret Key para validar informação que trocam entre si.
  - Um mecanismo MAC que se baseia em operações de hash criptográfico chama-se HMAC. Este tipo de MAC pode ser utilizado em conjunto com qualquer algoritmo de hash e.g. MD5 ou SHA-1, em combinação com uma chave secreta.
- Acordo de Chaves
  - Protocolo através do qual duas ou mais entidades conseguem estabelecer as mesmas chaves criptográficas sem trocarem informação secreta.

A classe Cipher

- A classe **cipher** constitui o núcleo da JCE, fornecendo mecanismos criptográficos de cifragem e decifragem de dados.
- Como todas as outras Engine Classes esta classe oferece o método getInstance. Aqui, em vez de um algoritmo, este método aceita o nome de uma *transformação*.
- Uma transformação é uma string que define quais as operações que devem ser efectuadas sobre os dados.
- O seu formato é *algorithm/mode/padding* i.e. além do algoritmo criptográfico, pode estar especificado um modo de funcionamento e um esquema de padding. Se estes últimos não são especificados, são utilizados valores por defeito.

Cipher: Modos de Funcionamento

- ECB (default) (Electronic Code Book) – Cifra blocos de dados em blocos cifrados, de forma independente. Muitas vezes utilizado para cifrar chaves.
- CBC (Cipher Block Chaining) – Introduz segurança adicional na forma de um mecanismo de feedback entre blocos: a decifragem de um bloco só é possível se o anterior for conhecido na sua versão cifrada. Muitas vezes utilizado para a cifragem de ficheiros.
- PCBC (Propagated Cipher Block Chaining) – Variação do anterior em que o feedback é estendido para incluir também a versão decifrada do bloco anterior.

Modos de Funcionamento (cont)

- CFB (Cipher Feedback) – Permite utilizar uma cifra de blocos em modo stream: a cifragem é feita bloco a bloco através de um XOR entre uma chave e o bloco de dados. A sequência (stream) de chaves é gerada pelo algoritmo de cifra por blocos, com base no ultimo bloco cifrado.
- OFB (Output Feedback) – Tal como o anterior, utiliza o algoritmo de cifra por blocos para gerar uma stream de chaves. No entanto, os dados não estão implicados na geração das chaves: apenas a ultima chave serve como base à geração da seguinte.
- Para estes 2 últimos modos é possível adicionar um número ao código do modo, com o tamanho das chaves.

Cipher: Inicialização

- Depois de se obter uma instancia de Cipher, é necessário chamar o método **init**, através do qual se indica a operação que se vai realizar:
  - ENCRYPT_MODE – Cifragem de dados.
  - DECRYPT_MODE – Decifragem de dados.
  - WRAP_MODE – Transformar uma chave num conjunto de bytes para que ela possa ser transportada de forma segura.
  - UNWRAP_MODE – Realizar a operação inversa da anterior, de forma a obter um objecto do tipo java.security.Key, a partir de um conjunto de bytes previamente obtidos em WRAP_MODE.
- Existem diversas versões do método **init**, que permitem fornecer parâmetros de inicialização, nomeadamente a chave, de diversas formas: uma chave, um certificado ou uma estrutura AlgorithmParameters.

Cipher: Operações

- Os dados podem ser cifrados ou decifrados de uma só vez (**doFinal**), ou por partes (**update**) o que pode ser útil se os dados não são todos conhecidos inicialmente.
- Quando a cifragem é efectuada por partes, o método **doFinal** tem de ser chamado para finalizar o processo e reinicializar o objecto para outras operações.
- Os métodos **wrap** e **unwrap** permitem transformar uma chave num conjunto de bytes que podem ser transferidos de forma segura, e vice-versa. Para se efectuar um unwrap, é preciso saber o algoritmo da chave (**Key.getAlgorithm**) e o seu tipo (**Cipher.PUBLIC_KEY**, **Cipher_PRIVATE_KEY** ou **Cipher.SECRET_KEY**).

Cipher: Outros Métodos

- O método **getParameters** permite obter um objecto do tipo **AlgorithmParameters** com os parâmetros que foram fornecidos ou gerados pelo próprio objecto para o algoritmo correspondente.
- O método **getIV** permite obter o vector de inicialização (primeira chave) utilizado pelos algoritmos de cifra em modo stream.
- O método **getOutputSize** permite saber qual tamanho do buffer necessário para armazenar os dados gerados pelo objecto.

Password-Based Encryption

- A implementação de cifras to tipo Password-Based Encryption baseiam-se no standard PKCS#5.
- As operações de cifragem/decifragem efectuam-se utilizando objectos do tipo Cipher, contendo implementações do algoritmo “PBEWithMD5AndDES ”.
- Para se utilizar estes objectos é também necessário parametrizar o objecto Cipher com o número de iterações e o chamado “salt”. Esta parametrização efectua-se recorrendo a um objecto do tipo PBEParameterSpec.
- A geração de chaves secretas para este algoritmo faz-se com base num objecto do tipo PBEKeySpec, que cria a chave com base na password, e utilizando a SecretKeyFactory para obter uma SecretKey comum.

Cipher Streams

- A JCE introduz o conceito de streams seguros, associados a objectos de cifragem/decifragem.
- As classes **CipherInputStream** e **CipherOutputStream** desempenham estas funções associando de forma transparente a operação de leitura/escrita num stream com a cifragem/decifragem dos dados.
- Estas classes seguem a semântica das classes stream do pacote **java.io,** implementando exactamente os mesmos métodos**,** e o seu comportamento ao nível das excepções é também semelhante.
- Internamente, estes objectos contêm um objecto do tipo **Cipher** e um objecto do tipo **Filter**[**Input**/**Output**]**Stream**.

Cipher Streams (cont)

- Os construtores destas classes aceitam como parâmetros um stream do tipo apropriado (i.e. input ou output, conforme o caso) e um objecto do tipo Cipher.
- O objecto do tipo Cipher tem de ser convenientemente inicializado, e de forma compatível com a função que vai desempenhar dentro do Cipher Stream.
- Note-se que não há uma associação Input Stream/Decifragem e Output Stream/Cifragem, uma vez que isto nos impediria, por exemplo, de abrir um ficheiro com um Cipher Input Stream e carrega-lo directamente para memória efectuando, simultaneamente, a sua cifragem.

SealedObject

- Esta classe permite criar um objecto e proteger a sua confidencialidade com um algoritmo criptográfico.
- Dado qualquer objecto que implemente a interface de serialização **java.io.Serializable** é possível encapsular a forma serializada desse objecto dentro de um SealedObject que o "esconderá" numa forma cifrada.
- O conteúdo protegido pode depois ser decifrado e o processo de serialização poderá então ser invertido para obter o objecto original.
- O encapsulamento consegue-se passando ao construtor da classe um objecto do tipo **Cipher**, convenientemente inicializado, e o objecto que se quer proteger.

SealedObject (cont)

- A recuperação do objecto original pode ser feita de duas formas:
  - Utilizando um objecto do tipo **Cipher**, correctamente inicializado, passando-o ao método **getObject** da classe SealedObject. Este método retornará o objecto original, caso a operação seja bem sucedida.
  - Utilizando a chave secreta apropriada, passando-a directamente ao método **getObject**. Neste caso, o objecto SealedObject cria ele próprio o objecto do tipo Cipher de que necessita, inicializando-o com a chave secreta que lhe é fornecida e com parâmetros adicionais que tenha armazenado internamente.

Mac

- A engine class **Mac** fornece a funcionalidade de um Message Authentication Code (MAC).
- Depois de obtida uma instância de um objecto deste tipo, ele deve ser inicializado utilizando o método **init**, que aceita como parâmetro a chave secreta a ser utilizada.
- Uma inicialização específica do algoritmo escolhido, com base em objectos do tipo **AlgorithmParameterSpec** pode também ser utilizada.
- A utilização de um objecto **Mac** é semelhante à do objecto **Cipher**, e baseia-se nos métodos **update** e **doFinal**, permitindo este último obter o Message Authentication Code pretendido.