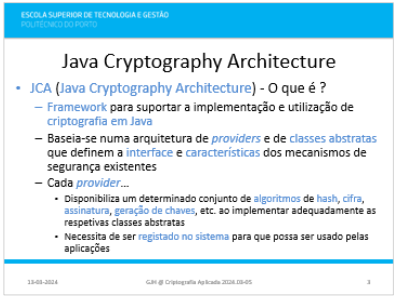
ESCOLA
SUPERIOR
DE TECNOLOGIA
E GESTÃO

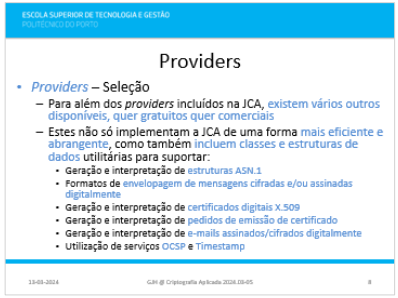
P.PORTO

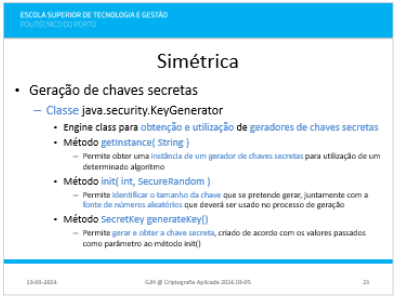
Criptografia Aplicada

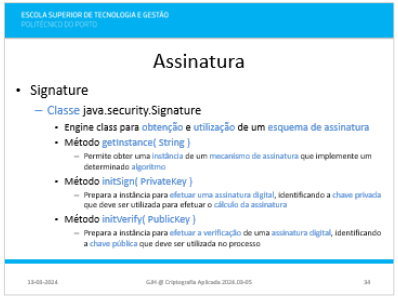
Criptografia em Java

Criptografia em Java









Java Cryptography Architecture

- JCA (Java Cryptography Architecture) - O que é ?
  - Framework para suportar a implementação e utilização de criptografia em Java
  - Baseia-se numa arquitetura de *providers* e de classes abstratas que definem a interface e características dos mecanismos de segurança existentes
  - Cada *provider*…
    - Disponibiliza um determinado conjunto de algoritmos de hash, cifra, assinatura, geração de chaves, etc. ao implementar adequadamente as respetivas classes abstratas
    - Necessita de ser registado no sistema para que possa ser usado pelas aplicações

Java Cryptography Architecture

- JCA (Java Cryptography Architecture) – Princípios
  - Independência em relação às implementações
    - Em vez de implementarem algoritmos de segurança, as aplicações pedem-nas a *providers* registados no sistema
  - Interoperabilidade de implementações
    - Ao assegurar que cada *provider* implementa as mesmas interfaces, permite-se que as aplicações sejam independentes do *provider* utilizado em cada momento/sistema (desde que incluam um conjunto mínimo de funcionalidades)
  - Extensibilidade de algoritmos
    - Aplicações podem estender as capacidades do sistema implementando os seus próprios *providers* e/ou algoritmos

Java Cryptography Architecture

- JCA (Java Cryptography Architecture) – História
  - Por razões históricas (relacionadas com restrições à exportação de criptografia dos EUA), inicialmente estava subdividida em dois componentes:
    - Java Security API
      - Classes java.security.*, distribuídas livremente com o JDK
      - Disponibilizavam algoritmos de hash e assinatura
    - JCE (Java Cryptography Engine)
      - Classes javax.crypto.*, não distribuídas fora dos EUA, embora existissem implementações independentes realizadas por instituições externas
      - Disponibilizavam algoritmos de cifra e troca de chaves

Java Cryptography Architecture

- JCA (Java Cryptography Architecture) – Atualidade
  - A JCE é distribuída em conjunto com o JDK[1]
    - JDK 9 e superior disponibiliza algoritmos criptográficos fortes por omissão
    - JDK 6, 7 e 8 nas versões anteriores a 6u181, 7u171 e 8u161 necessitam de Unlimited Strength Jurisdiction Policy disponível no website da Oracle
  - A JCA inclui alguns providers[2] base (Sun, SunRsaSign, SunJCE, etc.)
    - Disponibilizam implementações de referência de alguns algoritmos de hash, cifra, assinatura, geração de chaves, etc.
    - Não são apropriados para utilização em situações de elevada exigência, dado que apenas suportam alguns algoritmos e não têm preocupações de otimização de desempenho

1) Operação não é possível a partir de certos países (Cuba, Iraque, Síria, Irão, etc.)
2) Mais informações em http://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html

Java Cryptography Architecture

- JCA (Java Cryptography Architecture) – Documentação
  - http://download.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html
  - http://download.oracle.com/javase/8/docs/api/java/security/package-summary.html
  - http://download.oracle.com/javase/8/docs/api/javax/crypto/package-summary.html

Providers

- *Providers* – Seleção
  - Para além dos *providers* incluídos na JCA, existem vários outros disponíveis, quer gratuitos quer comerciais
  - Estes não só implementam a JCA de uma forma mais eficiente e abrangente, como também incluem classes e estruturas de dados utilitárias para suportar:
    - Geração e interpretação de estruturas ASN.1
    - Formatos de envelopagem de mensagens cifradas e/ou assinadas digitalmente
    - Geração e interpretação de certificados digitais X.509
    - Geração e interpretação de pedidos de emissão de certificado
    - Geração e interpretação de e-mails assinados/cifrados digitalmente
    - Utilização de serviços OCSP e Timestamp

Providers

- *Providers* – Seleção
  - Exemplos
    - BouncyCastle
      - Toolkit criptográfico open-source bastante completo, incluindo não só a implementação de bastantes algoritmos como também muitas classes utilitárias
      - Utilizado por diversas empresas nos componentes criptográficos
      - *Provider* implementado pela classe org.bouncycastle.jce.provider.BouncyCastleProvider
      - Documentação:
        - » http://www.bouncycastle.org/specifications.html
        - » http://www.bouncycastle.org/docs/docs1.5on/index.html
        - » http://www.bouncycastle.org/docs/pkixdocs1.5on/index.html

Providers

- *Providers* – Seleção
  - Exemplos
    - IAIK
      - Toolkit criptográfico comercial desenvolvido pelo Instituto de Tecnologia da Universidade de Graz (Áustria)
      - Mais completo do que o BouncyCastle
      - A sua utilização é paga (excepto para fins educacionais), sendo possível contratar um serviço de suporte
      - *Provider* implementado pela classe iaik.security.provider.IAIK
      - Documentação:
        - » http://javadoc.iaik.tugraz.at/iaik_jce/current/index.html

Providers

- *Providers* – Instalação
  - Passos Preparatórios
    - Descarregar o(s) ficheiro(s) JAR do provider
    - Colocar ficheiro(s) JAR no CLASSPATH da máquina virtual ou do projeto
    - Consultar a documentação do *provider* para identificar a sua classe principal (que estende a classe java.security.Provider)

Providers

- *Providers* – Instalação
  - Instalação estática
    - Efetuada através da alteração de um ficheiro de configuração da JRE (lib/security/java.security), acrescentando uma linha semelhante a esta:
      - security.provider.n=<identificação da classe principal>
    - onde n representa a prioridade[1] atribuída ao *provider* na procura da implementação de um determinado algoritmo
    - Tem a vantagem de ser transparente para as aplicações, que não necessitam de fazer nada para poder utilizar esse *provider*
    - Contudo, implica acesso físico por parte do utilizador/integrador ao sistema de ficheiros da máquina onde a aplicação será utilizada

1) Tanto mais alta quando mais baixo for o valor de n (cujo mínimo é 1)

Providers

- *Providers* – Instalação
  - Instalação estática
    - Exemplo

Ordem de procura de cada implementação genérica solicitada

```
security.provider.1=org.bouncycastle.jce.provider.BouncyCastleProvider
security.provider.2=sun.security.provider.Sun
security.provider.3=sun.security.rsa.SunRsaSign
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
```

Providers

- *Providers* – Instalação
  - Instalação dinâmica
    - Efetuada explicitamente pelas aplicações no seu código, utilizando um dos métodos da classe java.security.Security
    - Tem a vantagem de permitir às aplicações que utilizem um *provider* da sua preferência, mesmo que este não se encontrasse previamente instalado no sistema
    - Exemplos:
      - Security.addProvider(new BouncyCastleProvider());
        - » Adiciona o *provider* ao final da lista dos *providers* já registados
      - Security.insertProviderAt(new BouncyCastleProvider(), 1);
        - » Adiciona o *provider* a uma determinada posição (1 neste caso) da lista dos *providers* registados

Providers

- *Providers* – Utilização
  - Para que possam aceder a uma implementação de um determinado mecanismo de segurança, as aplicações necessitam de o obter da JCA, utilizando o método getInstance() de uma das *engine* classes existentes para o efeito
  - Esse pedido pode ser efetuado de forma…
    - Implícita
      - A aplicação não especifica o *provider* que pretende utilizar, identificando apenas o algoritmo desejado

```
MessageDigest md = MessageDigest.getInstance( "SHA256" );
```

      - Caso o mesmo algoritmo seja implementado por mais do que um *provider* registado no sistema, será devolvida a implementação do que tiver maior prioridade (n menor)

Providers

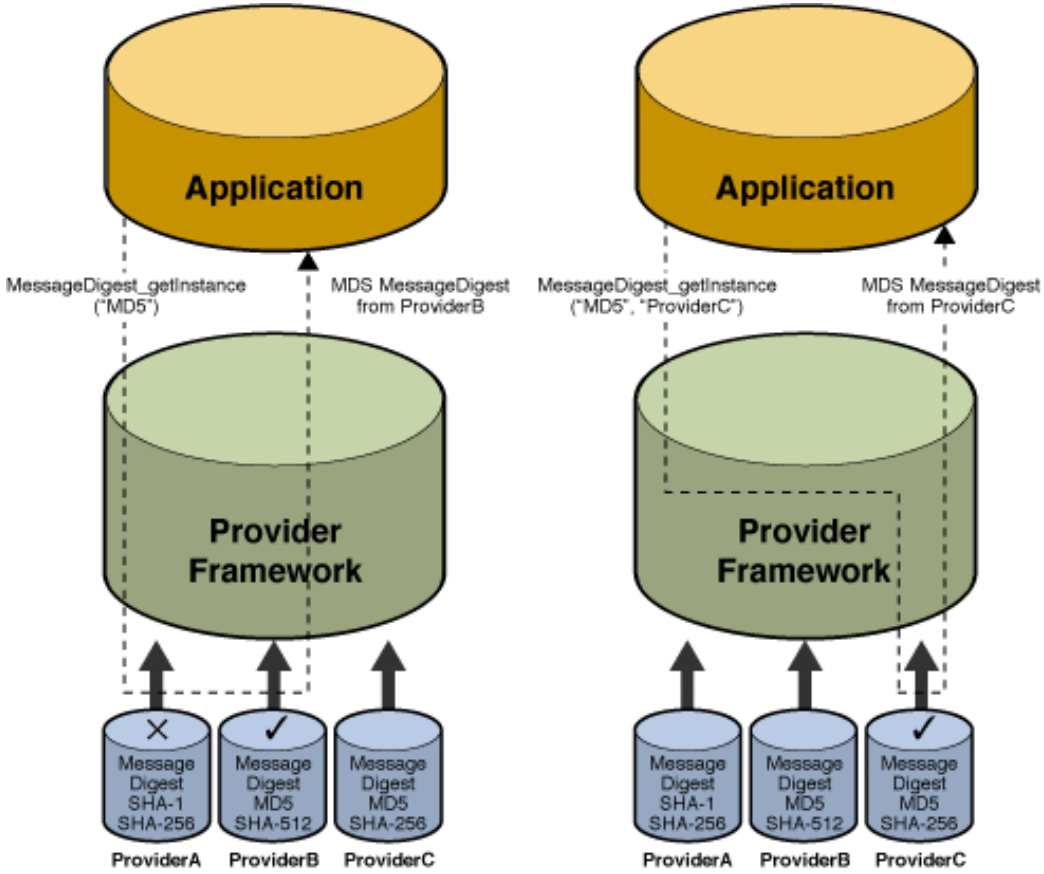
- *Providers* – Utilização
  - Esse pedido pode ser efetuado de forma…
    - Explícita
      - A aplicação especifica quer o algoritmo quer o *provider* que pretende utilizar

```
MessageDigest md = MessageDigest.getInstance( "SHA256", "BC" );
```

      - Apesar de contrariar o espírito subjacente à arquitetura JCA, em cenários de maior exigência/necessidade de segurança, pode ser mais apropriado por permitir manter o controlo sobre as implementação efetivamente utilizadas, evitando:
        - » As que não sejam consideradas seguras e/ou eficientes
        - » Que um atacante possa comprometer a segurança da aplicação ao injetar no sistema implementações adulteradas

Providers

- *Providers* – Utilização
  - Implícitos vs Explícitos



Utilitários

- Componentes utilitários da JCA
  - Classe java.security.Security
    - Permite…
      - Consultar e gerir os providers registados no sistema
      - Consultar e gerir as propriedades de segurança do sistema
  - Interface java.security.Key
    - Permite…
      - Definir uma interface comum para todo o tipo de chaves utilizadas nos mecanismos criptográficos da JCA
    - É estendida por interfaces mais específicas…
      - java.security.SecretKey para utilização em cifras simétricas
      - java.security.PrivateKey e java.security.PublicKey para utilização em cifras assimétricas

Utilitários

- Geração de números aleatórios
  - Classe java.security.SecureRandom
    - *Engine* class para obtenção e utilização de geradores de números aleatórios
    - Método getInstance(String)
      - Permite obter uma instância de um gerador de números aleatórios para utilização de um determinado algoritmo
    - Método setSeed(long)
      - Permite especificar o valor da semente do gerador de números aleatórios, que definirá a sequência de números gerada por este
      - Caso não seja invocado, o valor de semente será gerado pelo sistema
    - Método nextBytes(byte [ ])
      - Permite preencher o array, recebido como parâmetro, com números aleatórios

Utilitários

- Geração de números aleatórios
  - Classe java.security.SecureRandom
    - Exemplo:

```
SecureRandom sr = null;
try {
    // Criar uma fonte de números aleatórios segura baseada no
    // algoritmo "SHA1PRNG"
    sr = SecureRandom.getInstance( "SHA1PRNG" );
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace( System.err );
}
```

Caso não exista nenhum *provider* registado que disponibilize o algoritmo

Simétrica

- Geração de chaves secretas
  - Classe java.security.KeyGenerator
    - Engine class para obtenção e utilização de geradores de chaves secretas
    - Método getInstance(String)
      - Permite obter uma instância de um gerador de chaves secretas para utilização de um determinado algoritmo
    - Método init(int, SecureRandom)
      - Permite identificar o tamanho da chave que se pretende gerar, juntamente com a fonte de números aleatórios que deverá ser usado no processo de geração
    - Método SecretKey generateKey()
      - Permite gerar e obter a chave secreta, criado de acordo com os valores passados como parâmetro ao método init()

Simétrica

- Geração de chaves secretas
  - Classe javax.crypto.KeyGenerator - Exemplo:

```
KeyGenerator kGen = null;
SecretKey sk = null;

try {
    // Obter instância de gerador de chaves secretas "AES"
    kGen = KeyGenerator.getInstance( "AES" );
    // Configurar o gerador de chaves, definindo o tamanho da chave
    // desejada
    // NOTA: Para utilizar chaves superiores a 128 bits é necessário
    //       proceder à instalação da Java Cryptography Extension
    //       Unlimited Strength Jurisdiction Policy Files
    kGen.init( 128, sr );
    // Gerar chave secreta AES
    sk = kGen.generateKey();

} catch (NoSuchAlgorithmException e) {
    e.printStackTrace( System.err );
}
```

SecureRandom previamente inicializado

Caso não exista nenhum provider registado que disponibilize o algoritmo

Assimétrica

- Geração de par de chaves
  - Classe java.security.KeyPairGenerator
    - Engine class para obtenção e utilização de geradores de pares de chaves
    - Método getInstance(String)
      - Permite obter uma instância de um gerador de pares de chaves para utilização de um determinado algoritmo
    - Método initialize(int, SecureRandom)
      - Permite identificar o tamanho do par de chaves que se pretende gerar, juntamente com a fonte de números aleatórios que deverá ser usada no processo de geração
    - Método KeyPair generateKeyPair()
      - Permite gerar e obter um par de chaves, criado de acordo com os valores passados como parâmetro ao método initialize()

Assimétrica

- Geração de chaves secretas
  - Classe java.security.KeyPairGenerator - Exemplo

```
KeyPairGenerator kpGen = null;
KeyPair kp = null;

try {
    // Obter instância de gerador de pares de chaves "RSA"
    kpGen = KeyPairGenerator.getInstance( "RSA" );
    // Inicializar gerador de pares de chaves, definindo tamanho e
    // fonte de números aleatórios
    kpGen.initialize( 1024, sr );
    // Gerar par de chaves RSA
    kp = kpGen.generateKeyPair();

    System.out.println( "Key pair generated !" );

    System.out.println( "Private key details: " + kp.getPrivate().toString() );
    System.out.println( "Public key details: " + kp.getPublic().toString() );
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace( System.err );
}
```

SecureRandom previamente inicializado

Caso não exista nenhum provider registado que disponibilize o algoritmo

Simétrica / Assimétrica

- Cifragem / Decifragem
  - Classe java.security.Cipher
    - Engine class para obtenção e utilização de um mecanismo de cifragem/decifragem, simétrica ou assimétrica
    - Método getInstance(String)
      - Permite obter uma instância de um mecanismo de cifragem/decifragem que implemente um determinado…
        - » algoritmo de cifra
        - » modo de funcionamento da cifra (opcional)
        - » algoritmo de padding (opcional)
    - Exemplo:

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

Algoritmo, Modo, Padding

Simétrica / Assimétrica

- Cifragem / Decifragem
  - Classe java.security.Cipher
    - Método init (int opmode, Key key)
      - Permite inicializar a instância num dos modos (cifragem/decifragem) e indicar qual a chave a utilizar no processo
    - Exemplo:

      cipher.init (Cipher.ENCVRYPT_MODE, chave);

Simétrica / Assimétrica

- Cifragem / Decifragem
  - Classe java.security.Cipher
    - Método byte[ ] update(byte[ ])
      - Permite passar os bytes que se pretende cifrar/decifrar, podendo ser invocado tantas vezes quanto as necessárias
      - Devolve o resultado parcial da operação de cifragem/decifragem, pelo que o mesmo não deve ser descartado
    - Método byte[ ] doFinal(byte[ ])
      - Semelhante ao método update() mas apenas deve ser invocado para passar o último conjunto de bytes que se pretende cifrar/decifrar
      - Caso o conjunto total de bytes recebidos através de invocações do método update() e doFinal() não seja múltiplo do tamanho de bloco da cifra utilizada, este método irá aplicar o algoritmo de padding identificado aquando da obtenção da instância de Cipher

Simétrica / Assimétrica

- Cifragem / Decifragem - Exemplo

```
IvParameterSpec ivSpec = null;
Cipher cipher = null;
byte data[] = new byte[10];
byte encrypted[] = null;
// Declaração do vector de inicialização, que necessita de ter o tamanho de um bloco AES (16 bytes = 128 bits)
byte init_vector[] = new byte[16];

// Utilizar o gerador de números aleatórios para definir dados a cifrar
sr.nextBytes( data );
// Utilizar o gerador de números aleatórios para definir o vector de inicialização da cifra
sr.nextBytes( init_vector );
try {
    // Criar um contentor para o vector de inicialização da cifra
    ivSpec = new IvParameterSpec( init_vector );

    System.out.println("Data to encrypt: " + Utils.toHex(data, data.length) + " bytes: " + data.length);

    // Obter instância da cifra "AES", no modo de utilização "SIC" e com o padding "ISO10126-2"
    cipher = Cipher.getInstance( "AES/SIC/ISO10126-2Padding" );
    // Configurar a instância da cifra para o modo de cifragem, indicando a chave secreta a utilizar e o contentor do vector de
    // inicialização da cifra
    cipher.init( Cipher.ENCRYPT_MODE, sk, ivSpec );
    // Efectuar a cifragem
    encrypted = cipher.doFinal( data );

    System.out.println("Encrypted data: " + Utils.toHex(encrypted, encrypted.length) + " bytes: " + encrypted.length);
}
```

SecureRandom previamente inicializado

SecretKey previamente inicializada

Funções de Hash

- Funções de Hash
  - Classe java.security.MessageDigest
    - Engine class para obtenção e utilização de funções de hash
    - Método getInstance(String)
      - Permite obter uma instância de uma função de hash que implemente um determinado algoritmo
    - Método update(byte[ ])
      - Permite passar os bytes sobre os quais se pretende calcular o hash
      - Pode ser chamado várias vezes, sendo o hash calculado sobre a concatenação de todos os valores passados como parâmetro a este método
    - Método byte[ ] digest(byte[ ])
      - Permite obter o valor de hash, calculado sobre a concatenação dos valores passados como parâmetro aos métodos update() e digest()

Funções de Hash

- Funções de Hash
  - Classe java.security.MessageDigest
    - Exemplo

```
MessageDigest md = null;
byte data[] = new byte[10];
byte hash[] = null;

// Utilizar o gerador de números aleatórios para definir dados de input
sr.nextBytes( data );
try {
    System.out.println("Data to protect: " + Utils.toHex(data, data.length) + " bytes: " + data.length);

    // Obter instância da função de hash "SHA256"
    md = MessageDigest.getInstance( "SHA256" );
    // Efectuar o cálculo do hash
    hash = md.digest( data );

    System.out.println("Generated Hash: " + Utils.toHex(hash, hash.length) + " bytes: " + hash.length);
}
```

Message Authentication Code

- MAC
  - Classe java.crypto.Mac
    - Engine class para obtenção e utilização de códigos de autenticação de mensagens (MAC)
    - Método getInstance(String)
      - Permite obter uma instância de um código de autenticação de mensagens que implemente um determinado algoritmo
    - Método init(Key)
      - Permite identificar a chave que deve ser utilizada no cálculo do MAC
      - A chave passada como parâmetro deve ser uma instância de SecretKey cujo algoritmo seja compatível com o mecanismo de MAC que está a ser utilizado
      - Caso se utilize o mecanismo HMAC, o nome do algoritmo é ignorado

Message Authentication Code

- MAC
  - Classe java.crypto.Mac
    - Engine class para obtenção e utilização de códigos de autenticação de mensagens (MAC)
    - Método update(byte[])
      - Permite passar os bytes da mensagem que se pretende autenticar
      - Pode ser chamado várias vezes, sendo o MAC calculado sobre a concatenação de todos os valores passados como parâmetro a este método
    - Método byte[] doFinal(byte[])
      - Semelhante ao método update() mas apenas deve ser invocado para passar o último conjunto de bytes da mensagem que se pretende autenticar
      - Devolve o valor do MAC, calculado sobre a concatenação dos valores passados como parâmetro aos métodos update() e doFinal()

Message Authentication Code

- MAC
  - Classe java.crypto.Mac
    - Exemplo

```
Mac mm = null;
byte data[] = new byte[10];
byte mac[] = null;

// Utilizar o gerador de números aleatórios para definir dados a autenticar
sr.nextBytes( data );
try {
    System.out.println("Data to authenticate: " + Utils.toHex(data, data.length) + " bytes: " + data.length);

    // Obter instância de um código de autenticação de mensagens "HMAC"
    // baseado na utilização do algoritmo de hash "SHA-256"
    mm = Mac.getInstance( "HMac-SHA256" );
    // Configurar a instância, indicando a chave secreta a utilizar
    mm.init( sk );
    // Obter o MAC
    mac = mm.doFinal( data );

    System.out.println("Generated MAC: " + Utils.toHex(mac, mac.length) + " bytes: " + mac.length);
}
catch( InvalidKeyException e ) {
    e.printStackTrace( System.err );
}
catch( NoSuchAlgorithmException e ) {
    e.printStackTrace( System.err );
}
```

Assinatura

- Signature
  - Classe java.security.Signature
    - Engine class para obtenção e utilização de um esquema de assinatura
    - Método getInstance(String)
      - Permite obter uma instância de um mecanismo de assinatura que implemente um determinado algoritmo
    - Método initSign(PrivateKey)
      - Prepara a instância para efetuar uma assinatura digital, identificando a chave privada que deve ser utilizada para efetuar o cálculo da assinatura
    - Método initVerify(PublicKey)
      - Prepara a instância para efetuar a verificação de uma assinatura digital, identificando a chave pública que deve ser utilizada no processo

Assinatura

- Signature
  - Classe java.security.Signature
    - Método update(byte[ ])
      - Permite passar os bytes da mensagem que se pretende assinar (caso tenha sido previamente invocado o método initSign()) ou verificar (caso tenha sido previamente invocado o método initVerify())
      - Pode ser chamado várias vezes, sendo a mensagem final a assinar/verificar composta pela concatenação de todos os valores passados como parâmetro a este método
    - Método byte[ ] sign()
      - Calcula e devolve o valor da assinatura digital dos valores passados como parâmetro através do método update(), calculada usando a chave passada como parâmetro ao método initSign()

Assinatura

- Signature
  - Classe java.security.Signature
    - Método boolean verify (byte[ ])
      - Utiliza a chave pública facultada ao método initVerify() para verificar se o valor recebido como parâmetro corresponde à assinatura digital dos valores passados como parâmetro através do método update()

Assinatura

- Signature
  - Classe java.security.Signature
    - Exemplo Assinatura

```
Signature sm = null;
byte data[] = new byte[10];
byte signature[] = null;

// Utilizar o gerador de números aleatórios para definir dados a assinar
sr.nextBytes(data);
try {
    System.out.println("Data to sign: " + Utils.toHex(data, data.length) + " bytes: " + data.length);

    // Obter instância do esquema de assinatura "RSA", combinado com a
    // função de hash "SHA-256"
    sm = Signature.getInstance( "SHA256withRSA" );
    // Configurar a instância do esquema de assinatura para o modo de
    // assinatura, indicando a chave privada a utilizar
    sm.initSign( kp.getPrivate() );
    // Passar dados a assinar
    sm.update( data );
    // Calcular e obter a assinatura
    signature = sm.sign();

    System.out.println("Signature: " + Utils.toHex(signature, signature.length) + " bytes: " + signature.length);
}
```

Assinatura

- Signature
  - Classe java.security.Signature
    - Exemplo Verificação

```
boolean result;

try {
    System.out.println("Signed data: " + Utils.toHex(data, data.length) + " bytes: " + data.length);
    System.out.println("Signature: " + Utils.toHex(signature, signature.length) + " bytes: " + signature.length);

    // Obter instância do esquema de assinatura "RSA", combinado com a
    // função de hash "SHA-256"
    sm = Signature.getInstance( "SHA256withRSA" );
    // Configurar a instância do esquema de assinatura para o modo de
    // verificação, indicando a chave pública a utilizar
    sm.initVerify( kp.getPublic() );
    // Passar dados assinados
    sm.update( data );
    // Calcular resultado da verificação
    result = sm.verify( signature );

    System.out.println("Signature verification Result: " + result);
}
```