Sichere Programmierung
Praktikum 4
Kryptographie mit Java

Nils Klein
(79373)

Alexander Krause
(79878)

Abdullah Yildiz
(79669)

Dozentin: Corina Hampel

Inhaltsverzeichnis

1 Teil 1: Kryptographische Hashfunktion

1.1 Ablauf und Erklärung

Durch die Klasse MessageDigest wird für ein Text eine Prüfsumme nach SHA-256 berechnet. Die berechnete Prüfsumme wird anschließend mithilfe einer Methode in Hexadezimale Zahlen konvertiert. Zusätzlich wird die Prüfsumme mit einem Offset von zwei Formatiert. Das bedeutet, nach jedem zwei'ten Buchstaben wird ein Leerzeichen eingefügt. Die letzte Funktion soll eine Überprüfung eines Textes und der Prüfsumme durchführen. Hierfür wird ein Text und eine bekannte Prüfsumme benötigt. Für den übergebenen Text berechnet das Programm die Prüfsumme aus, und prüft ob es der übergebenen Prüfsumme entspricht.

1.2 Methode - sha256()

Um eine Prüfsumme zu berechnen, muss ein MessageDigest Objekt (hier "md") erstellt und initialisiert werden. Die Initialisierung geschieht im öffentlichen Konstruktor mit dem Initialwert "null".

```
1 this.md = null;
```

Das Objekt **md** weiß zunächst noch nicht welchen Algorithmus er verwenden soll. Durch **MessageDigest.getInstance("")** kann der gewünschte Algorithmus verwendet werden. Es sind z.B. folgende Algorithmen verwendbar: MD5, SHA-1, SHA-256. In unserem Fall wird SHA-256 benötigt. Die Instanz von **md** muss in einem Try-Catch-Block stattfinden. Es könnte sonst zu einem Fehler führen der das Programm abstürzen lässt. Speziell muss der Fall abgefangen werden, falls der Algorithmus **SHA256** nicht verfügbar wäre:

```
1 try {
2     this.md = MessageDigest.getInstance("SHA-256");
3 } catch (NoSuchAlgorithmException e){
4     System.out.println("Algorithmus SHA-256 nicht vorhanden!");
5 }
```

1.3 Methode - toHexString(byte[] data , int offset)

Die Methode **toHexString** erwartet zwei Übergabeparameter. Der erste Parameter ist die zuvor berechnete Prüfsumme in einem Byte-Array. Der Zweite Parameter ist optional, bei Verwendung wir entsprechend nach einem Offset ein Leerzeichen eingefügt. Es ist somit nur für die Textformatierung zuständig.
In Zeile 5 wird ein StringBuilder Objekt erstellt. An dieses Objekt können Strings "angehangen" werden.

```
1 public String toHexString(byte[] data , int offset) {
2     if (offset < 0) {
3         offset = 0;
4     }
5     StringBuilder sb = new StringBuilder ();
```

Nun müssen alle Bytes in eine Hexadezimale Zahl umgewandelt, und an das Objekt **sd** angehangen werden. Dies geschieht durch eine for-Schleife. Je nach Offset wird ein Leerzeichen nach einer bestimmten Anzahl an die Hexadezimale Zeichen (hier z.B. nach zwei Zeichen) eingefügt.

```
1 for (int i=0; i<data.length; i++) {
2     sb.append(String.format("%02X", data[i]));
3     if ((offset >0) && (i+1<data.length) &&
4                                     ((i+1) % offset == 0)) {
5         sb.append(" ");
6     }
7  }
```

Am ende wird das Ergebnis in die lokale Variable **checksum** abgespeichert. Der Rückgabeparameter ist die Prüfsumme bestehend aus Hexadezimalwerten.

```
1 this.checksum = sb.toString();
2 return sb.toString(); //Ausgabe der Hexadezimalen Prüfsumme
```

1.4 Methode - checksum(String text, String b_checksum)

Um eine Überprüfung der Prüfsumme vorzunehmen, wird ein **Text** und die dazugehörige **Prüfsumme** vorausgesetzt.
Als erstes berechnet die Funktion für den übergebenen **Text** eine Prüfsumme. Diese Prüfsumme nennen wir **a_checksum**:

```
public boolean checksum(String text, String b_checksum) {
    byte[] digest = this.md.digest(text.getBytes());
    String a_checksum = toHexString(digest, 2);
```

Da wir mit der Übergebenen Prüfsumme **(b_checksum)** nun **zwei** Prüfsummen haben, können diese miteinander verglichen werden. Sind die Prüfsummen gleich wird "true" zurückgegeben. Andernfalls "false".

```
    if(a_checksum.equals(b_checksum)) {
        System.out.println
           ("Wort entspricht der Übergebenen Prüfsumme: True");
        return true;
    }
    System.out.println
        ("Wort entspricht nicht der Übergebenen Prüfsumme: False");
    return false;
}
```

1.5 Methode - main(String[] args)

In der Main Methode wird die Instanz der Klasse erstellt, dadurch können die Methoden innerhalb der Klasse verwendet werden.

```
sha256 sha256 = new sha256();
```

Durch den aufruf **shas256.hilfsfunc(message)** wird der Text: "**Kryptographie macht Spass!!!**" übergeben und verschlüsselt.

```
sha256.hilfsfunc(message);
```

Durch den Aufruf **shas256.checksum(message, checksum)** wird der Text:
"**Kryptographie macht Spass!!!**" mit der dazugehörigen Prüfsumme übergeben und überprüft.

```
sha256.checksum(message, checksum);
```

Wir haben die Nachrichten fest (statisch) in unserem Programm hinterlegt. Jedoch könnte die Eingabe des Textes auch von einem User-Input erfolgen.

2 Teil 2: Symmetrische Verschlüsselung

2.1 Ablauf und Erklärung

Die symmetrische Verschlüsselung kann durch die Klasse **Cipher** realisiert werden. Es werden **Schlüssel**, **Initialisierungsvektor** (für den CBC-Mode) und ein **Text zum verschlüsseln/entschlüsseln** benötigt. Durch die Klassen SecureRandom und SecretKeySpec kann ein Schlüssel, auch Key genannt, generiert und spezifiziert werden. Für den CBC-Mode der AES-Verschlüsselung wird zusätzlich die Klasse **IvParameterSpec** benötigt. Der Initalisierungsvektor wird durch die Zufallszahlen des **SecureRandom** und mit **IvParameterSpec** spezifiziert. Schlussendlich kann damit ver/entschlüsselt werden.

2.2 Methode - encrypt(String p_text, String key)

Um einen Text Symmetrisch zu verschlüsseln, wird der zu **verschlüsselnde Text** und ein **Schlüssel** (Key) benötigt.

```
1 public String encrypt(String p_text, String key){
```

Der **Initialisierungsvektor** wird mit dem Objekt von SecretKeySpec **"keySpec"** und einem 16-Byte-Array zufällig erzeugt.

```
1 try{
2     SecureRandom random = new SecureRandom ();
3     byte[] iv = new byte[16];
4 
5     random.nextBytes(iv);
6     IvParameterSpec ivSpec = new IvParameterSpec(iv);
7 
8     SecretKeySpec keySpec =
9                     new SecretKeySpec(key.getBytes (), "AES");
```

Dem Cipher Objekt muss jetzt eine Instanz zugewiesen werden (mögliche Instanzen: ECB, CBC, CFB, OFB, CTR).

```
1     //Zuweisung der Instanz (CBC = "AES/CBC/PKCS5Padding")
2     Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
```

Im Anschluss wird das **Cipher** Objekt mit folgenden Werten initialisiert:

```
1 cipher.init(Cipher.ENCRYPT_MODE , keySpec , ivSpec);
```

Da wir verschlüsseln möchten wird **ENCRYPT_MODE** (Verschlüsselungsmodus) verwendet. Als nächstes ist nun die Verschlüsselung des Klartextes dran. Durch folgenden Befehl wird der Klartext in Bytes umgewandelt und in ein Byte-Array gespeichert:

```
1 byte[] cipherText = cipher.doFinal(p_text.getBytes());
```

Nun muss der Initalisierungsvektor zusammen mit dem Geheimtext in einem Byte-Array gespeichert und als Base64 String kodiert werden. Hierfür wird ein ByteArrayOutputStream Objekt **"os"** erstellt.

```
ByteArrayOutputStream os = new ByteArrayOutputStream ();
```

Die Bytes werden nun in das Objekt **"os"** geschrieben:

```
os.write(iv);
os.write(cipherText);
```

Damit wir eine Codierung durch Base64 möglich ist, muss der encoder initalisiert werden. Im Anschluss wird dem encoder das StringByte **"os"** mit der Funktion **encodeToString()** übergeben und als neuen String gespeichert.

```
Base64.Encoder encoder = Base64.getEncoder ();
String message = encoder.encodeToString(os.toByteArray ());
```

Die erste Ausgabe zeigt den Klartext an. Die zweite Ausgabe gibt den verschlüsselten Text wieder.

```
System.out.println("Plain text.: " + p_text);
System.out.println("Message....: " + message);
```

Der Rückgabewert der Methode ist die nun **verschlüsselte Nachricht**.

```
return message;
```

Falls hierbei etwas schief gehen sollte, wird durch den try-catch-Block die Fehlermeldung abgefangen, eine Ausgabe ausgegeben und durch **"return null"** das Programm beendet.

```
} catch (NoSuchAlgorithmException | NoSuchPaddingException |
    InvalidKeyException | InvalidAlgorithmParameterException |
    IllegalBlockSizeException | BadPaddingException |
                                            IOException e) {
    System.out.println("Fehler beim verschlüsseln!");
    return null;
}
```

2.3 Methode - decrypt(String encrypted, String key)

Um eine Entschlüsselung eines verschlüsselten Textes durchzuführen, wird der **Verschlüsselte Text**, und der dazu **verwendete Schlüssel** benötigt.

```
1 public String decrypt(String encrypted, String key) {
```

Auch hier wird zunächst der Schlüssel erstellt mit **SecretKeySpec**.

```
1 SecretKeySpec keySpec =  new SecretKeySpec(key.getBytes(),"AES");
```

Im Anschluss wird der **Base64.Decoder** initialisiert und ein Byte-Array der Länge 16 benötigt.

```
1 Base64.Decoder decoder = Base64.getDecoder();
2 byte[] iv = new byte[16];
```

Mithilfe der **decode** Methode wird der verschlüsselte Text dekodiert.

```
1 byte[] dec = decoder.decode(encrypted);
```

Um nun alle Bytes in den Initialisierungsvektor zu schreiben, wird eine for-Schleife verwendet. Die Schleife soll jedes Feld des Arrays **iv** mit den Werten von **dec** beschreiben (16 Felder).

```
1 for(int i=0; i < 16; i++) {
2     iv[i] = dec[i];
3 }
```

Im Anschluss wird dem Cipher Objekt die Instanz des CBC-Mode übergeben: ”AES/CBC/PKCS5Padding”.

```
1 IvParameterSpec ivSpec = new IvParameterSpec(iv);
2 Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
3 cipher.init(Cipher.DECRYPT_MODE, keySpec, ivSpec);
```

Da wir entschlüsseln möchten, wird der **DECRYPT_MODE** (Entschlüsselungsmodus) verwendet.
Der entschlüsselte Text wird dem Cipher Objekt mit der Funktion **do.Final** übergeben. Der Text wird nun in einer Aktion in einem Byte-Array gespeichert.

```
1 byte[] plain = cipher.doFinal(dec);
```

Das Array wird als nächstes in einen String konvertiert, eine Ausgabe erfolgt und wird als Rückgabeparameter zurückgegeben.

```
1 String org_text = new String(plain).substring(16);
2 System.out.println("Original Text: " + org_text);
3 return org_text;
```

Falls hierbei etwas schief gehen sollte, wird durch den try-catch-Block die Fehlermeldung abgefangen, eine Ausgabe ausgegeben und durch **"return null"** das Programm beendet.

```
} catch (NoSuchAlgorithmException | NoSuchPaddingException |
         InvalidKeyException |
         InvalidAlgorithmParameterException |
         IllegalBlockSizeException |
         BadPaddingException e) {
    System.out.println("Fehler beim entschlüsseln!");
    return null;
}
```

2.4 Methode - generate_Key()

Um einen Zufallsschlüssel zu generieren, wird die Klasse **SecureRandom** verwendet. Es wird zunächst ein neues Objekt **random** und ein **16-Byte-Array** erstellt.

```
SecureRandom random = new SecureRandom();
byte[] bytes = new byte[16];
```

Durch **next.Bytes** werden die Zufallszahlen von **random** in das Array geschrieben.

```
random.nextBytes(bytes);
```

Um nun die Werte verwenden zu können, müssen die Werte in einen String konvertiert werden.

```
String new_key = new String(bytes);
```

Nach der Konvertierung wird der String zurückgegeben, und wir haben einen Zufallsschlüssel generiert.

```
return new_key;
```

2.5 Methode - main(String[] args)

In der Main Methode wird die Instanz der Klasse erstellt, dadurch können die Methoden innerhalb der Klasse verwendet werden.

```
aes aes = new aes();
```

Als Schlüssel wird hier ein zufällig Generierter Schlüssel verwendet.

```
String random_key = aes.generate_Key();
```

Um nun zu verschlüsseln, muss die Methode mit einem Schlüssel und einem Text aufgerufen werden. Als Text wird die lokale deklarierte Variable **plainText** verwendet.

```
private static String plainText =
                    "Dies ist eine streng geheime Nachricht!!!";
String encryptedString = aes.encrypt(plainText, random_key);
```

Für die Entschlüsslung wird die Prüfsumme und der zum verschlüsseln verwendete Schlüssel verwendet.

```
aes.decrypt(encryptedString, random_key);
```

3 Teil 3: Asymmetrische Verschlüsselung

3.1 Erklären der Verschlüsselung

Im Unterschied zur symmetrischen Verschlüsselung, wird bei der asymmetrischen Verschlüsselung mit unterschiedlichen Schlüsseln ver- bzw. entschlüsselt.
Möchte beispielsweise Alice eine verschlüsselte Nachricht an Bob senden, benötigt Alice den öffentlichen Schlüssel von Bob. Mit diesem verschlüsselt sie dann den Geheimtext und sendet ihn an Bob.
Bob kann diesen Text dann mit seinem privaten Schlüssel, der auf seinem Client verbleibt entschlüsseln und lesen.
Um also eine asymmetrische Verschlüsselung selbst durchzuführen, müssen wir uns ein solches Schlüsselpaar (bestehend aus öffentlichem und privatem Schlüssel, nachfolgend **pubKey** und **privKey** genannt) erzeugen.

3.2 Implementierung - Konstruktor

Die Klasse rsa.java besitzt zwei Objektvariablen, um pubKey sowie privKey nach Erzeugung abzuspeichern.
Außerdem haben wir den zu verschlüsselnden String "**Kryptographie macht immer noch Spass!!!**" hardcoded als Klassenkonstante abgespeichert, dies wäre auch dynamisch als User-Eingabe denkbar.

```
1 public rsa() throws NoSuchAlgorithmException
```

Der Konstruktor der Klasse erhält keine Übergabeparameter und wirft eine **NoSuchAlgorithmException** (wird von der getInstance() Methode der Klasse KeyPairGenerator geworfen).

Wir setzen hier zunächst die Schlüssellänge für das RSA Verfahren auf die vom BSI empfohlene Schlüssellänge. Bis Ende 2022 sind das 2000 Bit.

```
1     int keysize = 2000;
```

Nachfolgend wird ein Objekt der Klasse KeyPairGenerator erstellt und die Instanz auf RSA gesetzt. Diese Klasse benötigen wir, um das RSA-Schlüsselpaar zu generieren.

```
1     KeyPairGenerator keyPairGen;
2     keyPairGen = KeyPairGenerator.getInstance("RSA");
```

Der Schlüsselpaargenerator wird mit der vorher spezifizierten Schlüssellänge initialisiert und es wird ein KeyPair erzeugt.

```
1     keyPairGen.initialize(keysize);
2     KeyPair rsaKeyPair = keyPairGen.generateKeyPair();
```

Die beiden Schlüssel aus dem KeyPair speichern wir entsprechend in den Objektvariablen publicKey und privateKey.

```
1     this.publicKey = rsaKeyPair.getPublic();
2     this.privateKey = rsaKeyPair.getPrivate();
```

3.3 Methode - encrypt

Nachdem nun ein Objekt der Klasse rsa erstellt wurde, kann auf diesem Objekt die eigentliche Verschlüsselung mit der Methode **encrypt** durchgeführt werden.

```
1     public String encrypt(String plainText)
2         throws NoSuchPaddingException,
3             NoSuchAlgorithmException,
4             InvalidKeyException,
5             BadPaddingException,
6             IllegalBlockSizeException
```

Übergabeparameter: String, der verschlüsselt werden soll
Rückgabeparameter: Verschlüsselter String

Die diversen Exceptions werden von den einzelnen Methoden innerhalb von **encrypt** geworfen und hängen deswegen am Methodenkopf.

Nun wird ein Objekt ”rsa” der Klasse Cipher erstellt und der Modus dieses Objekts auf RSA gesetzt.

```
1     Cipher rsa;
2     rsa = Cipher.getInstance("RSA");
```

Der Cipher wird nun mit dem vorher erzeugten pubKey im Verschlüsselungsmodus (ENCRYPT_MODE) initialisiert.

```
1     rsa.init(Cipher.ENCRYPT_MODE, publicKey);
```

Wir definieren uns ein leeres byte-Array ”cipherText”. Anschließend führen wir die eigentliche Verschlüsselung durch, indem wir die Methode doFinal() mit dem zu verschlüsselnden String in Byteform aufrufen und den Rückgabewert der Methode in ”cipherText” speichern.

```
1     byte[] cipherText;
2     cipherText = rsa.doFinal(plainText.getBytes());
```

Der verschlüsselte Text in byte-Form wird abschließend noch Base64 kodiert und dann an die aufrufende Methode zurückgegeben.

```
1     Base64.Encoder encoder = Base64.getEncoder();
2     return encoder.encodeToString(cipherText);
```

3.4 Methode - decrypt

Ein RSA-verschlüsselter Text kann nun mit der gleichen Instanz mit der verschlüsselt wurde (andernfalls passt der key nicht) wieder entschlüsselt werden und zwar mit Hilfe der decrypt Methode.

```
1     private String decrypt(String encryptedText)
2         throws NoSuchAlgorithmException,
3             NoSuchPaddingException,
4             InvalidKeyException,
5             IllegalBlockSizeException,
6             BadPaddingException
```

Übergabeparameter: String, der entschlüsselt werden soll
Rückgabeparameter: Entschlüsselter String

Die diversen Exceptions werden von den einzelnen Methoden innerhalb von **decrypt** geworfen und hängen deswegen am Methodenkopf.

Nun wird ein Objekt ”rsa” der Klasse Cipher erstellt und der Modus dieses Objekts auf RSA gesetzt.

```
1     Cipher rsa;
2     rsa = Cipher.getInstance("RSA");
```

Da der verschlüsselte Text gemäß Vorgabe in Base64-Kodierung vorliegt, muss dieser zunächst dekodiert und in einem byte-Array ”cipherText” gespeichert werden.

```
1     Base64.Decoder decoder = Base64.getDecoder();
2     byte[] cipherText = decoder.decode(encryptedText);
```

Der Cipher wird nun mit dem vorher erzeugten privKey im Entschlüsselungsmodus (DECRYPT_MODE) initialisiert.

```
1     rsa.init(Cipher.DECRYPT_MODE, privateKey);
```

Abschließend führen wir die eigentliche Entschlüsselung durch, indem wir die Methode doFinal() mit dem zu entschlüsselnden String in Byteform aufrufen und den Rückgabewert der Methode in ”cipherText” speichern.

```
1     cipherText = rsa.doFinal(cipherText);
```

Mit diesem byte-Array wird ein neuer String erzeugt, der jetzt die entschlüsselte Nachricht enthält, und an die aufrufende Methode zurückgegeben.

```
1     return new String(cipherText);
```

3.5 Methode - main

Um die Funktionalität des Programms zu testen, bzw. den eigentlichen Verschlüsselungsmodus anschaulich darzustellen haben wir eine ausführbare main-Methode hinzugefügt.

In dieser erzeugen wir ein Objekt der Klasse rsa, rufen also den Konstruktor auf und generieren somit unser keyPair.

```
1     rsa rsa = new rsa();
```

Auf der Kommandozeile werden nacheinander der zu verschlüsselnde String (abgespeichert als Klassenkonstante "plainText"), der verschlüsselte plainText und der entschlüsselte cipherText ausgegeben.

```
1     System.out.println("Plain Text:    " + plainText);
2     System.out.println("Verschlüsselt: " + rsa.encrypt(plainText));
3     System.out.println("Entschlüsselt: "
4         + rsa.decrypt(rsa.encrypt(plainText)));
```

Um gerade bei längeren Strings nicht händisch überprüfen zu müssen, ob der "rsa-verarbeitete" String gleich dem originalen plainText ist, haben wir noch eine .equals() Abfrage eingebaut, die bei erfolgreichem Ausführen **true** ausgibt.

```
1     System.out.print("plainText und verschlüsselter
2         -> entschlüsselter Text gleich : ");
3     System.out.println(plainText.equals(
4             rsa.decrypt(rsa.encrypt(plainText))
5         ));
```

4 Teil 4: Digitale Signatur

4.1 Erklären des Signaturvorgangs

Folgendes Szenario: Alice möchte Bob eine Nachricht schicken. Bob möchte überprüfen können, ob die Nachricht auf dem Weg verfälscht, ausgetauscht oder beschädigt wurde. Wie machen wir das ?
Die Lösung lautet: Alice muss die Nachricht signieren.
Dafür muss Alice von der Nachricht einen Hash erstellen und diesen mit ihrem privateKey verschlüsseln. Alice schickt dann diesen verschlüsselten Hash zusammen mit Ihrer Nachricht an Bob.
Bob entschlüsselt nun den hash mit Alice' öffentlichem Schlüssel, verifiziert also somit, dass die Nachricht von ihr kommt (Nur Alice verfügt über ihren privaten Schlüssel).
Anschließend hasht Bob über die Nachricht und vergleicht den entstandenen Hash mit dem erhaltenen, entschlüsselten. Sind beide hashes gleich, weiß Bob, dass die Nachricht nicht verändert oder beschädigt wurde.

4.2 Implementierung - Konstruktor

Da wir in diesem Beispiel mit Hilfe des RSA Verfahrens signieren sollen, haben wir uns dazu entschlossen, die Klasse **SHA256withRSA** als Unterklasse von **rsa.java** zu implementieren.
Aus diesem Grund ist unser Konstruktor auch vergleichsweise klein. Wir rufen hier lediglich den Konstruktor der Oberklasse auf um das Schlüsselpaar zu generieren und setzen den Signaturmodus des Signatur Objekts "rsa" auf "SHA256withRSA". Das Signaturobjekt wird als Instanzvariable abgespeichert.
NoSuchAlgorithmException kann von **.getInstance()** geworfen werden und ist deswegen wieder im Methodenkopf deklariert.

```
1     public SHA256withRSA() throws NoSuchAlgorithmException {
2         super();
3         rsa = Signature.getInstance("SHA256withRSA");
4     }
```

4.3 Methode - sign

Die Methode **sign()** führt die Signatur eines Strings durch.

```
1     public String sign(String plainText)
2         throws SignatureException,
3         InvalidKeyException
```

Übergabeparamter: String, der signiert werden soll
Rückgabeparameter: Verschlüsselter Hash über den String

Die diversen Exceptions werden von den einzelnen Methoden innerhalb von **sign** geworfen und hängen deswegen am Methodenkopf.

Als erstes wird nun der vorher erzeugte privKey der Oberklasse an ”rsa” übergeben und damit dem Objekt mitgeteilt, dass es mit diesem privKey signieren soll.

```
1     rsa.initSign(super.privateKey);
```

Anschließend erstellen wir ein leeres byte-Array, in dem der Rückgabewert der **rsa.sign()** Methode gespeichert werden soll. Mit der Methode **rsa.update()** übergeben wir den zu signierenden String in Byteform an das Signaturobjekt ”rsa”.
rsa.sign führt nun die eigentliche Signatur durch und gibt uns den hash in Byteform zurück, welchen wir in ”signature” abspeichern.

```
1     byte[] signature;
2     rsa.update(plainText.getBytes());
3     signature = rsa.sign();
```

Dieses byte-Array wird abschließend gemäß Vorgabe als Base64 kodierter String an die aufrufende Methode zurückgegeben.

```
1     Base64.Encoder encoder = Base64.getEncoder();
2     return encoder.encodeToString(signature);
```

4.4 Methode - verify

Die Methode **verify** führt die Verifikation einer Signatur eines Strings durch.

```
1     public boolean verify(String plainText, String signature)
2             throws InvalidKeyException, SignatureException
```

Übergabeparameter **plainText**: String, der signiert wurde
Übergabeparameter **signature**: Signatur, die überprüft werden soll Rückgabeparameter: Angabe, ob die Signatur für den übergebenen String gültig ist

Die diversen Exceptions werden von den einzelnen Methoden innerhalb von **verify** geworfen und hängen deswegen am Methodenkopf.

Als erstes wird der Base64-kodierte Signatur-hash (siehe Vorgabe) dekodiert und in einem byte-Array "decSignature" gespeichert.

```
1     Base64.Decoder decoder = Base64.getDecoder();
2     byte[] decSignature = decoder.decode(signature);
```

Anschließend wird der vorher erzeugte pubKey der Oberklasse an "rsa" übergeben und damit dem Objekt mitgeteilt, dass es mit diesem pubKey die Verifizierung der Signatur initialisieren soll.

```
1     rsa.initVerify(super.publicKey);
```

Mit der Methode **rsa.update()** übergeben wir den plainText String in Byteform an das Signaturobjekt "rsa".

```
1     rsa.update(plainText.getBytes());
```

rsa.verify führt nun die eigentliche Verifizierung durch und gibt uns true zurück, wenn die Verifizierung erfolgreich war, andernfalls false.
Diesen boolean geben wir abschließend an die aufrufende Methode zurück.

```
1     return rsa.verify(decSignature);
```

4.5 Methode - main

Um die Funktionalität des Programms zu testen, bzw. den eigentlichen Signaturvorgang anschaulich darzustellen haben wir eine ausführbare main-Methode hinzugefügt.

In dieser erzeugen wir ein Objekt der Klasse SHA256withRSA, rufen also den Konstruktor auf, generieren somit unser keyPair und setzen den Modus auf ”SHA256withRSA” (siehe 4.2).

```
1     SHA256withRSA sha = new SHA256withRSA();
```

Auf der Kommandozeile werden nacheinander der zu signierende String (abgespeichert als Klassenkonstante ”plainText”), die Signatur von plainText und der boolean, ob die Verifizierung erfolgreich war ausgegeben.

```
1     System.out.println("Plain Text:     " + plainText);
2     System.out.println("Signatur:      " + sha.sign(plainText));
3     System.out.println("Verifiziert:   "
4         + sha.verify(plainText, sha.sign(plainText)));
```