**DIGITAL SIGNATURE SYSTEM IMPLEMENTED USING RSA ALGORITHM**

**Two different programs**

I will present and explain two different programs in this lesson.  The first program named **Rsa09** will illustrate the digital signing of an *unencrypted* message using an encrypted digital signature based on an SHA-1 message digest using RSA algorithm.

The second program named **Rsa10** will illustrate the digital signing of an *encrypted* message using an encrypted digital signature based on an SHA-1 message digest using RSA algorithm.

**Three important aspects**

There are at least three important aspects of electronic communication:

* **Authentication:** Confirming the identities of the parties involved.
* **Confidentiality:** Making certain that only authorized parties can understand the message, even if it is intercepted by unauthorized persons.
* **Integrity:** Confirming that the content of the message wasn't modified during transmission.

The first program named **Rsa09** illustrates a scenario where only *authentication* and *integrity* are of concern.  In this scenario, there is no need to be concerned about the public disclosure of the content of a message.

The second program named **Rsa10** illustrates a scenario where all three aspects are important.  In this case, it is also necessary to keep the content of the message confidential.

**Discussion and Sample Code**

**The program named Rsa09**

IMPORTANT: BECAUSE THIS PROGRAM USES PREDETERMINED KEYS THAT WERE DESIGNED FOR A BLOCK SIZE OF 12, IT WILL WORK CORRECTLY ONLY FOR A BLOCK SIZE OF 12.

**The purpose of the program**

The purpose of this program is to illustrate the signing of an unencrypted message using an encrypted message digest as the digital signature, along with the later verification of the signature by the receiver of the message.

**Alice and Bob**

Many of the comments in this program reflect the scenario of Alice and Bob, which is common in discussions of RSA.  Alice needs to send a *signed* unencrypted message to Bob.  The message needs to be signed so that Bob can be confident that the message that he received was actually sent by Alice and was not sent by someone pretending to be Alice.

Bob also needs to confirm the integrity of the message by confirming that the message was not modified during the transmission from Alice to Bob.

**Creating a digital signature**

Alice creates a digital signature by getting an SHA-1 message digest for her message.  Then she encodes and encrypts the digest using her private key and the RSA algorithm.  The encrypted digest becomes her digital signature.  She signs the message by appending the encrypted digest to the end of the message.  Bob can later decrypt the digital signature using Alice's public key.

**Separating the message from the signature**

Bob will need some way to separate the message from the digital signature.  Therefore, before sending the signed message to Bob, Alice appends four numeric characters to the end that specify the length of the original message in characters.  Then Alice sends the signed message to Bob.

Bob extracts the message length from the end of the signed message.  He uses this information to separate the original message from the digital signature.  Then he decrypts and decodes the digital signature using Alice's public key.

**What does Bob already know?**

Bob knows that the original message digest was 40 hexadecimal characters in length, because an SHA-1 message digest is always 160 bits or 20 bytes in length.  Two hexadecimal characters are required to represent each byte, resulting in 40 hexadecimal characters for a hexadecimal representation of an SHA-1message digest.

Bob also knows that any additional characters beyond the first 40 hexadecimal characters were the result of extending the message digest to make it possible to encrypt it using the RSA encryption algorithm.  Bob discards all but the first 40 characters from the decoded digital signature.

**Checking the message digest**

Then Bob computes an SHA-1 message digest for the extracted message and compares it with the first 40 characters of the decoded digital signature.  If they match, he concludes that the digital signature is valid.  If not, he concludes that the digital signature is invalid.

If the digital signature is valid, Bob has confirmed both the authenticity and the integrity of the message from Alice.  It could only have been sent by Alice *(or by someone having access to her private key).*  Furthermore, the message could not have been modified after Alice computed her message digest and created her digital signature.

This program allows the message to contain all of the ASCII characters from space (32) through ~ (126) inclusive.

**Predetermined keys**

This program uses predetermined values for the following:

* Alice's public key, e
* Alice's private key, d
* Alice's modulus operand, n

**Testing**

This program was tested using Sun's JDK 1.5 (BlueJ as java editor) and Win7.

**The program output**

This program produces the output shown in Figure 1. I will refer back to Figure 1 as I explain the code in this program.

|  |
| --- |
| 1. Alice's keys:  2. e: 17  3. d: 279263220413  4. n: 951386374109  5. Block size: 12  6. Alice's msg: Hello Bob, how are you? I am goi  ng to go to the movie tonight. I plan to see Gone  with the Wind. I really like Clark Gable. Woul  d you like to go with me? I would really like to  have some company. Your friend, Alice.  7. Alice's msg digest: 89CA87110E2063FAF4A364B77C  18C3260AD47B8E  8. Alice's extended msg digest: 89CA87110E2063FAF  4A364B77C18C3260AD47B8E00000000  9. Alice's encrypted digital signature: 511335051  5586806264156901277970003516182249744328267808527  91138100851742238493708680633911102408  10. Alice's signed msg: Hello Bob, how are you?  I am going to go to the movie tonight. I plan to  see Gone with the Wind. I really like Clark Gabl  e. Would you like to go with me? I would really  like to have some company. Your friend, Alice.5  1133505155868062641569012779700035161822497443282  6780852791138100851742238493708680633911102408  11. Alice's signed msg with msg length appended:  Hello Bob, how are you? I am going to go to the  movie tonight. I plan to see Gone with the Wind.  I really like Clark Gable. Would you like to go  with me? I would really like to have somecompan  y. Your friend, Alice.51133505155868062641569012  7797000351618224974432826780852791138100851742238  4937086806339111024080220  12. Bob's calculated msg len: 220  13. Bob's extracted msg text: Hello Bob, how are  you? I am going to go to the movie tonight. I pl  an to see Gone with the Wind. I really like Clar  k Gable. Would you like to go with me? I would  really like to have some company. Your friend, A  lice.  14. Bob's extracted extended digital signature: 5  1133505155868062641569012779700035161822497443282  6780852791138100851742238493708680633911102408  15. Bob's decoded extended digital signature: 89C  A87110E2063FAF4A364B77C18C3260AD47B8E00000000  16. Bob's decoded digital signature: 89CA87110E20  63FAF4A364B77C18C3260AD47B8E  17. Bob's digest: 89CA87110E2063FAF4A364B77C18C32  60AD47B8E  18. Bob's conclusion: Valid signature    **Figure 1** |

**Discuss in fragments**

I will discuss and explain this program by breaking it down into fragments and discussing the fragments.  You can view the program in its entirety in Listing 26 near the end of the lesson.

**The Rsa09 and Keys classes**

The class named **Rsa09** begins in Listing 1.  This class begins with the definition of a **static** class named **Keys**.

|  |
| --- |
| class Rsa09{  static class Keys{  BigInteger n =  new BigInteger("951386374109");  BigInteger d =  new BigInteger("279263220413");  BigInteger e = new BigInteger("17");  }//end inner class Keys  **Listing 1** |

**Purpose of the Keys class**

The **Keys** class is a static class whose sole purpose is to create an object that serves as a container for the keys.  It was made static so that it can be instantiated and accessed from within the **main** method.  An object of the **Keys** class contains precomputed values for the two keys, **e** and **d**, and for the modulus operand **n**.  In this context, **e** is the public key and **d** is the private key.

Note that the precomputed key values encapsulated in a **Keys** object were designed to be used with a block size of 12.

**Beginning of the main method**

The **main** method begins in Listing 2.

|  |
| --- |
| public static void main(String[] args){  Keys aliceKeys = new Keys();  **Listing 2** |

The code in Listing 2 instantiates an object of the **Keys** class, and refers to that object as **aliceKeys**.  The contents of the object represent Alice's public and private keys.

**The test message**

Listing 3 constructs a **String** object referred to as **aliceMsg**.  This **String** object constitutes the message that Alice will sign and send to Bob.

|  |
| --- |
| String aliceMsg = "Hello Bob, how are you? I "  + "am going to go to the movie tonight. I "  + "plan to see Gone with the Wind. I "  + "really like Clark Gable. Would you "  + "like to go with me? I would really "  + "like to have some company. Your "  + "friend, Alice.";  **Listing 3** |

**Miscellaneous operations**

The code in Listing 4 performs several miscellaneous operations.

|  |
| --- |
| //Set blockSize.  int blockSize = 12;  //Instantiate an object of this class  Rsa09 obj = new Rsa09();  //Display the key values along with the  // modulus operand.  System.out.println("1. Alice's keys:");  System.out.println("2. e: " + aliceKeys.e);  System.out.println("3. d: " + aliceKeys.d);  System.out.println("4. n: " + aliceKeys.n);  System.out.println("\n5. Block size: "  + blockSize);    //Display the message that Alice will sign  // and send to Bob.  System.out.println("\n6. Alice's msg: "  + aliceMsg);  **Listing 4** |

Listing 4 begins by setting the encryption block size to 12 in order to match the required block size for the keys shown in Listing 1.  Then Listing 4 instantiates an object of the **Rsa09** class and refers to that object as **obj**.

Following this, Listing 4 displays the text material identified by Line 1 through Line 6 in Figure 1.

**Get the message digest**

Listing 5 invokes the **digestIt** method to get a message digest that represents the **String** message referred to by **aliceMsg**.

|  |
| --- |
| byte[] aliceDigest =  obj.digestIt(aliceMsg.getBytes());  **Listing 5** |

The **digestIt** method generates and returns a digest for an incoming array of bytes using Sun's SHA message digest algorithm.

**Display the message digest in hexadecimal**

Listing 6 invokes the **byteArrayToHexStr** method to convert the message digest to a string of hexadecimal characters.

|  |
| --- |
| System.out.println(  "\n7. Alice's msg digest: "  + obj.byteArrayToHexStr(aliceDigest));  **Listing 6** |

Listing 6 also displays the hexadecimal representation of the message digest as Line number 7 in Figure 1.

The **byteArrayToHexStr** method converts an incoming array of bytes into a **String** that represents each of the bytes as two hexadecimal characters.

**Extend to multiple of block size**

RSA encryption algorithm is a block-oriented algorithm.  Therefore, the length of the material to be encrypted must be a multiple of the block size.

Listing 7 extends the length of the digest to force this condition to be satisfied.

|  |
| --- |
| int tailLen = aliceDigest.length % blockSize;  int extendLen = 0;  if((tailLen > 0)){  extendLen = blockSize - tailLen;  }//end if  //Create an array for the extended digest  // with all elements initialized to a value  // of 0.  byte[] aliceExtendedDigest =  new byte[aliceDigest.length + extendLen];  //Copy the digest into the bottom of the new  // array. The empty portion at the top of  // the array becomes the extension.  System.arraycopy(aliceDigest,0,  aliceExtendedDigest,0,  aliceDigest.length);  **Listing 7** |

**Calculate the required length**

Listing 7 begins by calculating the required length of the extended message digest.

**Create array object of required length**

Then Listing 7 creates an array object of type **byte** with a **length** equal to the required extended length of the message digest.  Recall that all array elements are initialized to a value of 0 for type **byte**.

**Populate the array**

Then Listing 7 copies the message digest into the lower elements of the array object.  The elements at the top with a default value of 0 constitute the extension to the message digest.

**Convert the extended message digest to hexadecimal**

Listing 8 invokes the **byteArrayToHexStr** method to convert the extended message digest into hexadecimal format.

|  |
| --- |
| String aliceExtendedDigestAsHex =  obj.byteArrayToHexStr(aliceExtendedDigest);  System.out.println(  "\n8. Alice's extended msg digest: "  + aliceExtendedDigestAsHex);  **Listing 8** |

Listing 8 also displays the extended message digest in hexadecimal format producing Line number 8 in the output shown in Figure 1.  As you can see from Figure 1, for a block size of 12, it was necessary to extend the hexadecimal representation of the message digest from 40 to 48 hexadecimal characters.

*(Bob will discard the final eight characters later because he knows that only 40 hexadecimal characters are required to represent a message digest that is computed using the SHA-1 algorithm.)*

**Encode the extended message digest**

As you will recall from earlier lessons, the input to the RSA algorithm must be in the form of integer numeric data.  Therefore, Listing 9 invokes the **encode** method to encode the hexadecimal representation of the extended message digest into integer numeric format in preparation for encryption.

|  |
| --- |
| String aliceEncodedDigest =  obj.encode(aliceExtendedDigestAsHex);  **Listing 9** |

**The encode method**

The purpose of the **encode** method is to encode a plain text message into numeric format where:

* space = 32 - 32 = 0
* ...
* A = 65 - 32 = 33
* ...
* Z = 90 - 32 = 58
* ...
* a = 97 - 32 = 65
* ...
* ~ = 126 - 32 = 94

This is the place where the restriction on the allowable characters contained in the message or message digest comes into play.  A message digest expressed in hexadecimal format satisfies this restriction.

**Create the digital signature**

Alice creates the digital signature for her message by invoking the **doRSA** method to encrypt the encoded and extended message digest using her *private* key.  This is shown in Listing 10.  Bob can later decrypt Alice's digital signature for this message using Alice's *public* key.

|  |
| --- |
| String aliceSignature = obj.doRSA(  aliceEncodedDigest,aliceKeys.d,aliceKeys.n,  blockSize);  System.out.println(  "\n9. Alice's encrypted digital signature: "  + aliceSignature);  **Listing 10** |

Listing 10 also displays the encrypted digital signature producing Line 9 in the output shown in Figure 1.  It is worth noting at this point that the encrypted digital signature consists of 96 characters.

**Digital signature is a constant length**

The length of the encrypted digital signature will always be 96 characters for the SHA-1 message digest algorithm, the RSA encryption algorithm with a block size of 12, and the encoding algorithm used in this program to convert the hexadecimal representation of the message digest to integer numeric format.  That will be true even if the length of the message represented by the message digest is extended to many thousands of characters.

Thus, one advantage of the use of message digests for digital signatures is that the digital signature is always relatively short regardless of the length of the message.  Therefore, for long messages, the use of message digests for digital signatures results in conservation of communication bandwidth *(at the expense of extra computational requirements).*

**The doRSA method**

The **doRSA** method applies the RSA encryption algorithm to an input string using a specified exponent, a specified modulus operator, and a specified block size, which are provided as input parameters.

This method can be used to encrypt or to decrypt the input string depending on whether the exponent is an encryption key or a decryption key.

**Sign the message**

At this point, Alice signs the message by appending the encrypted digital signature onto the end of the actual text message.  This is shown in Listing 10.

|  |
| --- |
| String aliceSignedMsg =  aliceMsg + aliceSignature;  System.out.println(  "\n10. Alice's signed msg: "  + aliceSignedMsg);  **Listing 11** |

Listing 10 also displays the signed message, producing Line 10 in the output shown in Figure 1.

**Append the length of the message**

When Bob receives the signed message, he will need to separate the message text from the digital signature.  Somehow, he must know where the message text ends and the digital signature begins.  There are several ways to accomplish this.  This is OK, except that it means that the underscore character cannot be used elsewhere in the message, *(unless you want to also get in involved in the use of escape sequences as are commonly used in programming source code).*

**A different approach**

This program takes a different approach.  Rather than dedicating a delimiter character, the code in Listing 12 appends four numeric characters onto the end of the signed message indicating the length of the original message.  These numeric characters contain leading zeros if necessary.  Bob will extract and use this information to determine the length of the text message, and to separate the text message from the digital signature.  As written here, this approach can accommodate message lengths up to 9999 characters.

|  |
| --- |
| String aliceMsgLenAsStr =  "" + aliceMsg.length();  //Confirm number of characters in the string.  if((aliceMsgLenAsStr.length() > 4)  || (aliceMsgLenAsStr.length() <= 0)){  System.out.println(  "Message length error.");  System.exit(0);  }//end if  //Prepend leading zeros if necessary  if(aliceMsgLenAsStr.length() == 1){  aliceMsgLenAsStr = "000"  + aliceMsgLenAsStr;  }else if(aliceMsgLenAsStr.length() == 2){  aliceMsgLenAsStr = "00" + aliceMsgLenAsStr;  }else if(aliceMsgLenAsStr.length() == 3){  aliceMsgLenAsStr = "0" + aliceMsgLenAsStr;  }else if(aliceMsgLenAsStr.length() == 1){  aliceMsgLenAsStr = "000"  + aliceMsgLenAsStr;  }//end else    //Append the message length string to the  // previously signed message.  aliceSignedMsg = aliceSignedMsg  + aliceMsgLenAsStr;  System.out.println(  "\n11. Alice's signed msg with msg length "  + "appended: "  + aliceSignedMsg);  **Listing 12** |

**The code is straightforward**

Although the code fragment in Listing 12 is rather long, there is nothing conceptually difficult about the code in Listing 12.  This code simply creates a four-character **String** containing characters that indicate the length of the original message.  This four-character **String** is appended onto the end of the signed message.

Listing 12 also displays the signed message with the number of characters appended onto the end, producing Line 11 in the output shown in Figure 1.  As you can see, the last four characters of Line 11 in Figure 1 indicate that the length of the original text message was 220 characters.

**Send the message to Bob**

At this point, Alice sends her signed message to Bob.  This is accomplished conceptually in Listing 13 by assigning the reference contained in **aliceSignedMsg** to the reference variable named **bobSignedMsg**.

|  |
| --- |
| String bobSignedMsg = aliceSignedMsg;  **Listing 13** |

At this point, Bob has the message and can begin the process of validating it.  Bob will need to extract the message length, the message text, and the digital signature from the signed message.

**Get the message length**

Bob begins by extracting the length of the text message from the end of the signed message as shown in Listing 14.  Hopefully, you will have no difficulty understanding the code in Listing 14.

|  |
| --- |
| int bobMsgLen = Integer.parseInt(  bobSignedMsg.substring(  bobSignedMsg.length() - 4));  System.out.println(  "\n12. Bob's calculated msg len: "  + bobMsgLen);  **Listing 14** |

Listing 14 also displays the message length extracted by Bob, producing Line 12 in Figure 1.  As you can see from Line 12, Bob concludes that the length of the text message is 220 characters.

**Extract the message text**

Next Bob uses the message length to extract the first 220 characters from the signed message as shown in Listing 15.  This is the text of the message that Alice sent to Bob.

|  |
| --- |
| String bobMsgText = bobSignedMsg.substring(  0,bobMsgLen);  System.out.println(  "\n13. Bob's extracted msg text: "  + bobMsgText);  **Listing 15** |

Listing 15 also displays the message text producing Line 13 in Figure 1.

**Authenticate the source and confirm the integrity**

However, at this point, Bob can't be certain that the message was actually sent by Alice.  It may have been sent by someone pretending to be Alice.  Even if the message was sent by Alice, it may have been modified in transit.  Alice may not be going to see *Gone with the Wind* after all.  Her message may have been modified in transit, and she may actually be going to a hockey game instead of a movie.

Bob needs to use the digital signature to authenticate the source of the message and to confirm the integrity of the message.

**Extract the encoded and encrypted digital signature**

Bob knows that everything following the message text except for the four characters at the end *(that indicate the message length)* is the encoded and encrypted version of the extended digital signature.  He extracts that signature in Listing 16.

|  |
| --- |
| String bobExtractedSignature =  bobSignedMsg.substring(  bobMsgLen,bobSignedMsg.length() - 4);  System.out.println(  "\n14. Bob's extracted extended digital "  + "signature: "  + bobExtractedSignature);  **Listing 16** |

Listing 16 also displays this information, producing Line 14 in Figure 1.

**Decrypt and decode the extended digital signature**

At this point, Bob uses Alice's public key to decrypt the digital signature.  Then he invokes the **decode** method to decode the extended digital signature as shown in Listing 17.

|  |
| --- |
| //Bob decrypts the extended digital  // signature using Alice's public key.  String bobDecryptedExtendedSignature =  obj.doRSA(bobExtractedSignature,  aliceKeys.e,aliceKeys.n,blockSize);  //Bob decodes the extended digital signature.  String bobDecodedExtendedSignature =  obj.decode(  bobDecryptedExtendedSignature);  System.out.println(  "\n15. Bob's decoded extended digital "  + "signature: "  + bobDecodedExtendedSignature);  **Listing 17** |

**The output**

Listing 17 also displays the decoded digital signature, producing the output shown by Line 15 in Figure 1.  At this point, you can compare Line 15 with Line 8 in Figure 1 to confirm that they match.  Of course, Bob can't see Line 8 in Figure 1, so he has some more work to do.

**Discard the extended characters**

Bob knows that the digital signature produced by the SHA-1 message-digest algorithm consists of 40 hexadecimal characters.  He knows that any additional characters were put there simply to fill out the last block in order to make encryption possible using the RSA encryption algorithm.  He extracts the hexadecimal version of the digital signature by ignoring all but the first 40 hexadecimal characters as shown in Listing 18.

|  |
| --- |
| String bobDecodedSignature =  bobDecodedExtendedSignature.  substring(0,40);  System.out.println(  "\n16. Bob's decoded digital signature: "  + bobDecodedSignature);  **Listing 18** |

Listing 18 also displays the result, producing Line 16 in Figure 1.  You can compare Line 16 with Line 7 to confirm that they match.  Once again, however, Bob can't see Line 7 so he has some more work to do.

**Compute a message digest for comparison**

In order to authenticate the source of the message and to confirm the integrity of the message, Bob needs to use the same SHA-1 message-digest algorithm to compute a message digest for the received message and to compare it with the decrypted and decoded digital signature.

Bob does this in Listing 19.

|  |
| --- |
| //Bob computes the msg digest for comparison  // with the decoded signature.  byte[] bobDigest =  obj.digestIt(bobMsgText.getBytes());    System.out.println("\n17. Bob's digest: "  + obj.byteArrayToHexStr(bobDigest));  **Listing 19** |

**Compare the two message digests**

Listing 19 also displays the message digest produced by Bob and displays the result in Line 17 of Figure 1.  Since the message digest is only 40 hexadecimal characters in length, Bob could visually compare his version of the message digest shown in Line 17 with the decoded digital signature shown in Line 16 of Figure 1.  However, Bob decides to automate the comparison process as shown in Listing 20.

Listing 20 compares the decoded digital signature with Bob's computed message digest and displays an output message indicating whether or not the signature is valid.

|  |
| --- |
| if(bobDecodedSignature.equals(  obj.byteArrayToHexStr(bobDigest))){  System.out.println(  "\n18. Bob's conclusion: "  + "Valid signature");  }else{  System.out.println(  "\n18. Bob's conclusion: "  + "Invalid signature");  }//end else  }//end main  **Listing 20** |

For this particular case, Listing 20 produces the output shown in Line 18 in Figure 1.  Bob concludes that the digital signature contained in Alice's signed message is valid.

**Authenticity and integrity is confirmed**

Since the digital signature is valid, Bob can conclude that the message was sent by Alice *(or someone having access to her private key).*  Thus, the message has been *authenticated,* confirming the identity of the sender.

Bob can also conclude that the message was not modified by someone else during transmission *(unless that person had access to Alice's private key),* thus validating the integrity of the message.

On the other hand, if the digital signature had been deemed to be invalid, Bob would conclude that either the *authenticity* or the *integrity* of the message is questionable.

**The program named Rsa10 - a different scenario**

The purpose of the program named **Rsa10** is to illustrate the signing of an ***encrypted*** message along with the later verification of the signature by the receiver of the message.  This program differs from the previous program named **Rsa09** in that the previous program signed an ***unencrypted*** message.

As before, the comments in this program reflect the scenario of Alice and Bob.  However, this time, Alice needs to send a signed ***encrypted*** message to Bob instead of sending a signed ***unencrypted*** message to him.

**Authenticity and integrity**

As before, the message needs to be signed so that Bob can be confident that the message that he received was sent by Alice and not by someone pretending to be Alice.  Bob also needs to confirm the integrity of the message by confirming that the message was not modified during the transmission from Alice to Bob.

**Confidentiality**

Also, a very important aspect of this scenario is that the confidentiality of the message must be maintained.  That is, the message must be sent in such a way that it can only be read by Bob.

**Signing the message**

As before, Alice creates a digital signature by getting an SHA-1 message digest for her message.  Then she encodes and encrypts the digest using her *private* key and the RSA algorithm.  The encrypted digest becomes her digital signature.  Alice signs the message by appending the encrypted digest to the end of the message.  Bob can later decrypt the digital signature using Alice's *public* key.

**Maintaining confidentiality**

Up to this point, this program is very similar to the previous program, but this is where the two programs diverge.

Alice will ensure the confidentiality of the message by encrypting the entire signed message using Bob's *public* key.  Then only Bob *(or someone having access to Bob's private)* key can decrypt and read the message.

As before, Bob will need some way to separate the message from the digital signature after he decrypts and decodes the received message.  Alice will satisfy this need by appending numeric characters onto the end of the message to notify Bob of the length of the original message.  In addition, before encoding and encrypting the signed message using the RSA algorithm, Alice must extend the length of the signed message to cause the length of the signed and encoded message to be a multiple of the block size.

**Two birds with one stone**

Alice takes care of both of these requirements at the same time.  She extends the unencoded signed message with a pad that causes the extended length of the unencoded signed message to be a multiple of half the block size *(the encoding scheme in use doubles the length of the signed message, so the encoded message length will be a multiple of the block size).*

In addition, Alice causes the last four bytes in the pad to contain numeric characters specifying the length of the original message.  If the pad is more than four bytes long, Alice fills in the extra characters in the pad with equal characters (=).

*(Bob can later extract the length of the original message and use it to separate the message from the signature.  He can also discard the equal characters used to extend the length of the message.)*

**Send the message to Bob**

Then Alice sends the signed encoded and encrypted message to Bob.

**Decrypt and decode the signed message**

Bob begins by decrypting and decoding the encrypted signed message.  Then he extracts the length of the original message from the end of the decoded signed message.

**Separate the digital signature from the message**

Bob uses this information along with his knowledge that the pad may also contain equal characters (=) as filler, to extract the encrypted digital signature from the decoded signed message.  Bob also uses this information to extract the message from the decoded signed message.

**Decrypt and decode the digital signature**

At this point, this operations in this program have pretty well converged back into the operations in the previous program named **Rsa09**.  Bob decrypts and decodes the encrypted digital signature using Alice's public key.  As before, Bob knows that the original message digest was 40 hexadecimal characters in length, because an SHA-1 message digest is always 160 bits or 20 bytes in length.  Two hexadecimal characters are required to represent each byte, resulting in 40 hexadecimal characters for a hexadecimal representation of an SHA-1 message digest.

**Extract the message digest**

Bob also knows that any additional hexadecimal characters beyond the first 40 were the result of extending the message digest to make it possible to encrypt it using the RSA encryption algorithm.  Bob discards all but the first 40 characters from the decoded digital signature.

**Compute a new message digest for comparison**

Then Bob computes an SHA-1 message digest for the extracted message and compares it with the first 40 characters of the decoded digital signature.  If they match, he concludes that the digital signature is valid.  If not, he concludes that the digital signature is invalid.

**Confirming authenticity and integrity**

If the digital signature is valid, Bob has confirmed the authenticity and the integrity of the message from Alice.  It could only have been sent by Alice or by someone having access to her private key.  Furthermore, the message could not have been modified after Alice computed her message digest and created her digital signature.

**Maintaining confidentiality**

Finally, because Alice encrypted the signed message using Bob's public key, only Bob or someone having access to his private key can decrypt the signed message.  Therefore, the confidentiality of the message was protected during the transmission of the message from Alice to Bob.

**Allowable character set**

As before, this program allows all of the ASCII characters from space (32) through ~ (126) inclusive to be included in the message.  None of these characters are set aside to be used as delimiter characters.

*(Note that the use of the equal character (=) to extend the signed message prior to encoding and encryption does not prohibit the use of this character within the message itself.)*

**Predetermined keys**

This program uses predetermined values for the following:

* Alice's public key, e
* Alice's private key, d
* Alice's modulus operand, n
* Bob's public key, e
* Bob's private key, d
* Bob's modulus operand, n

The above keys were all designed to be used with a block size of 12.

**Discussion will be brief**

Much of the code in this program is identical to or very similar to the code in the previous program named **Rsa09**.  Because of the similarity of this program to the previous program, I will discuss only those code fragments that contain important differences between the two programs.  You can view the entire program in Listing 27 near the end of the lesson.

**The program output**

This program produces the output shown in Figure 2. I will refer back to Figure 2 while discussing the program code, and will comment only on those portions of the output that differentiate this program from the previous program.

|  |
| --- |
| 1. Alice's keys:  2. e: 17  3. d: 279263220413  4. n: 951386374109  5. Bob's keys:  6. e: 19  7. d: 799574694235  8. n: 951386374109  9. Block size: 12  10. Alice's msg: Hello Bob, how are you? I am go  ing to go to the movie tonight. I plan to see Gon  e with the Wind. I really like Clark Gable. Wou  ld you like to go with me? I would really like t  o have some company. Your friend, Alice.  11. Alice's msg digest: 89CA87110E2063FAF4A364B77  C18C3260AD47B8E  12. Alice's extended msg digest: 89CA87110E2063FA  F4A364B77C18C3260AD47B8E00000000  13. Alice's encrypted digital signature: 51133505  1558680626415690127797000351618224974432826780852  791138100851742238493708680633911102408  14. Alice's signed msg: Hello Bob, how are you?  I am going to go to the movie tonight. I plan to  see Gone with the Wind. I really like Clark Gabl  e. Would you like to go with me? I would really  like to have some company. Your friend, Alice.5  1133505155868062641569012779700035161822497443282  6780852791138100851742238493708680633911102408  15. Alice's padded signed msg with msg length app  ended: Hello Bob, how are you? I am going to go t  o the movie tonight. I plan to see Gone with the  Wind. I really like Clark Gable. Would you like  to go with me? I would really like to have some  company. Your friend, Alice.5113350515586806264  1569012779700035161822497443282678085279113810085  1742238493708680633911102408====0220  16. Alice's encrypted signed message: 91401046409  0477812185023522059255227103733444908183675749115  8834555666464747955317487974740038223011505609617  0958348608789466477424942329033371476785906742456  0777331120891074470078287368038504581398329728289  1169705147103312551664808525983850773644838883849  5892974479228714690946361944896047795453704351458  0138918238467231200275709874384289825441638463619  4489607764755499173131264107586650093298351648729  8983371264328395673083207830600854525558017959215  2887426943639881827352686383004770007421461800123  9893565435323832631156747443170942361455453339508  2810225070374501073338462893929842434027331818342  2870733493040858128661542384978031710885942868854  17. Bob's decrypted and decoded signed message: H  ello Bob, how are you? I am going to go to the m  ovie tonight. I plan to see Gone with the Wind.  I really like Clark Gable. Would you like to go  with me? I would really like to have some compan  y. Your friend, Alice.51133505155868062641569012  7797000351618224974432826780852791138100851742238  493708680633911102408====0220  18. Bob's calculated msg len: 220  19. Bob's extracted msg text: Hello Bob, how are  you? I am going to go to the movie tonight. I pl  an to see Gone with the Wind. I really like Clar  k Gable. Would you like to go with me? I would  really like to have some company. Your friend, A  lice.  20. Bob's extracted extended digital signature: 5  1133505155868062641569012779700035161822497443282  6780852791138100851742238493708680633911102408  21. Bob's decoded extended digital signature: 89C  A87110E2063FAF4A364B77C18C3260AD47B8E00000000  22. Bob's decoded digital signature: 89CA87110E20  63FAF4A364B77C18C3260AD47B8E  23. Bob's digest: 89CA87110E2063FAF4A364B77C18C32  60AD47B8E  24. Bob's conclusion: Valid signature    **Figure 2** |

**Instantiate two separate Keys objects**

The first important difference that we see occurs at the beginning of the **main** method shown in Listing 21.  This code instantiates and populates two objects of the class **Keys**.  One object is used to encapsulate predetermined keys for Alice.  The other object is used to encapsulate predetermined keys for Bob.

|  |
| --- |
| public static void main(String[] args){  //Instantiate an object containing  // Alice's keys.  Keys aliceKeys = new Keys();  aliceKeys.n = new BigInteger("951386374109");  aliceKeys.d = new BigInteger("279263220413");  aliceKeys.e = new BigInteger("17");    //Instantiate an object containing  // Bob's keys  Keys bobKeys = new Keys();  //Note, these key values were designed to  // be used with a block size of 12  bobKeys.n = new BigInteger("951386374109");  bobKeys.d = new BigInteger("799574694235");  bobKeys.e = new BigInteger("19");  **Listing 21** |

The contents of the two objects are later displayed producing Lines 1 through 8 in Figure 2.

**Skip code in the main method**

A large amount of code, which is either identical to, or very similar to code in the previous program named **Rsa09**, will be skipped between the code fragment in Listing 21 and the code fragment in Listing 22. Listing 22 picks up at the point where Alice has finished signing the message.  You can view the code that was skipped in Listing 27 near the end of the lesson.

**Prepare to encrypt the signed message**

Because Alice needs to maintain confidentiality, she will encrypt the signed message using Bob's public key before sending the message to him.  In preparation for encoding and encryption, she needs to extend the length of the signed message such that the total length is a multiple of half the block size.  *(The length of the encoded message will be a multiple of the block size.)*

In addition, Alice needs to notify Bob as to the length of the original message so that he will have a way to separate the text of the original message from the encrypted message signature.  She will accomplish both of these requirements at the same time.

Alice will extend the signed message using a pad that includes the length of the original message in the last four bytes of the pad.  All bytes in the pad other than the last four bytes will be filled with equal characters (=).

**Invoke the padTheMsg method**

Continuing with the **main** method, Alice accomplishes this by invoking the method named **padTheMsg** as shown in Listing 22.

|  |
| --- |
| String alicePaddedSignedMsg =  obj.**padTheMsg**(aliceSignedMsg,blockSize/2,  aliceMsg.length());  System.out.println(  "\n15. Alice's padded signed msg with msg "  + "length appended: "  + alicePaddedSignedMsg);  **Listing 22** |

Listing 22 also displays the padded signed message after the original message length has been appended as the last four characters in the pad.  This output is shown in Line 15 in Figure 2.  Line 15 shows that the original message contained 220 characters and also shows the equal characters used as filler in the pad.

**The method named padTheMsg**

Setting the **main** method aside temporarily, the method named **padTheMsg** is shown in its entirety in Listing 23.

|  |
| --- |
| private String padTheMsg(  String msgIn,int block,int origMsgLen){  byte[] msgData = msgIn.getBytes();  int msgInLen = msgData.length;  int tailLength = msgInLen%block;  int padLength = 0;  if((block - tailLength >= 4))  padLength = block - tailLength;  else  padLength = 2\*block - tailLength;    //Create a four-byte array containing  // characters that indicate the length of  // the original msg with leading zeros if  // necessary.  String msgLenAsStr = "" + origMsgLen;  //Confirm number of characters.  if((msgLenAsStr.length() > 4)  || (msgLenAsStr.length() <= 0)){  System.out.println(  "Message length error.");  System.exit(0);  }//end if    //Prepend leading zeros if necessary  if(msgLenAsStr.length() == 1){  msgLenAsStr = "000" + msgLenAsStr;  }else if(msgLenAsStr.length() == 2){  msgLenAsStr = "00" + msgLenAsStr;  }else if(msgLenAsStr.length() == 3){  msgLenAsStr = "0" + msgLenAsStr;  }else if(msgLenAsStr.length() == 1){  msgLenAsStr = "000" + msgLenAsStr;  }//end else    //Create a four-byte array containing the  // original message length as four numeric  // characters with leading zeros.  byte[] msgLenAsBytes =  msgLenAsStr.getBytes();    //Construct an array containing the bytes  // required to make the padded message length  // equal to a multiple of block  byte[] thePad = new byte[padLength];  //Populate the array with = characters.  // No need to put = characters in the last  // four bytes.  for(int cnt = 0;cnt < thePad.length - 4;  cnt++){  thePad[cnt] = '=';  }//end for loop    //Put the original message length at the end  // of the pad.  System.arraycopy(msgLenAsBytes,0,thePad,  thePad.length - 4,4);    //Create an output array.  byte[] output =  new byte[msgInLen + padLength];  //Populate the output array with the original  // msgData concatenated with the pad.  System.arraycopy(msgData,0,output,0,  msgInLen);  System.arraycopy(  thePad,0,output,msgInLen,thePad.length);    //Convert the output to a String and return  // the String..  String outputAsStr = new String(output);  return outputAsStr;  }//end padTheMsg  **Listing 23** |

The method named **padTheMsg** extends a message such that the final length is a multiple of the incoming parameter named **block** and the last four bytes specify the value of the incoming parameter named **origMsgLength**.  All the remaining bytes in the pad are filled with equal characters (=).  The padded message is returned as type **String**.

While the method is rather long, it is not conceptually difficult, and the comments should be self explanatory.  Therefore, I won't discuss the method in detail.

**Encode and encrypt the signed message**

Returning now to the **main** method, Alice invokes the **encode** and **doRSA** methods, *(which were discussed previously),* to first encode and then encrypt the signed message using Bob's public key.  This is shown in Listing 24.

|  |
| --- |
| //Encode the message.  String aliceEncodedSignedMsg =  obj.encode(alicePaddedSignedMsg);  //Encrypt the message.  String aliceEncryptedSignedMsg = obj.doRSA(  aliceEncodedSignedMsg,bobKeys.e,bobKeys.n,  blockSize);  System.out.println(  "\n16. Alice's encrypted signed message: "  + aliceEncryptedSignedMsg);  **Listing 24** |

Listing 24 also displays the encrypted message as Line 16 in Figure 2.

Then Alice sends the encrypted signed message to Bob.

**Decrypt and decode the message**

Bob invokes the **doRSA** and **decode** methods to decrypt and decode the message using his private key, as shown in Listing 25.

|  |
| --- |
| String bobEncryptedSignedMsg =  aliceEncryptedSignedMsg;    //At the receiving end, Bob begins by  // decrypting and decoding the signed  // message using his private key.  //Decrypt the message.  String bobDecryptedSignedMsg = obj.doRSA(  bobEncryptedSignedMsg,bobKeys.d,bobKeys.n,  blockSize);  //Decode the message.  String bobDecodedSignedMsg =  obj.decode(bobDecryptedSignedMsg);    System.out.println(  "\n17. Bob's decrypted and decoded signed "  + "message: "+ bobDecodedSignedMsg);  **Listing 25** |

Listing 25 also displays the decrypted and decoded message producing Line 17 in Figure 2.  As you can see, Line 17 in Figure 2 matches Line 15 in Figure 2, confirming that the encryption and later decryption process was successful.

**Confirming authenticity and integrity**

At this point, knowing the format of the signed message, Bob has all the information that he needs to confirm the authenticity and the integrity of the message from Alice.

The remaining code in the program named **Rsa10** is not identical to, but is very similar to the code used for the same purposes in the previous program named **Rsa09**.  This code, along with my comments in the code, should be self explanatory.  Therefore, I won't discuss it further.  You can view the remaining code in Listing 27 near the end of the lesson.

**Run the Programs**

I encourage you to copy, compile and run the following programs that are provided in this lesson:

* Rsa09
* Rsa10

Experiment with the programs, making changes and observing the results of your changes.

Above all, have fun and use these programs to learn as much as you can about the theory behind and the mechanics of digital signatures using message digests and public key cryptography as implemented using Java. Good luck to you Mama T.

**Summary**

There are numerous protocols that can be used along with public key cryptography and digital signatures to validate the authenticity and integrity of a message.

In an earlier lesson, I explained one of those protocols for the use of digital signatures.  For that protocol, the sender used her private key to encrypt the entire message and appended the encrypted message onto the original message as a digital signature.  That approach can be wasteful of communication bandwidth when the message is long.

In this lesson, the sender uses the SHA-1 algorithm to compute a fixed-length message digest of the original message and encrypt using RSA Agorithm.  The message digest is encrypted using the sender's private key and appended onto the original message as a digital signature.  This approach conserves communication bandwidth for long messages, but requires an extra computational step at both the sending and receiving ends.

This lesson applies the message-digest approach for two different scenarios.  In the first scenario, it was necessary to confirm the authenticity and integrity of a message, but there was no requirement to keep the contents of the message secret.  In the second scenario, it was necessary to keep the contents of the message secret in addition to confirming the authenticity and integrity of the message.  In the second scenario, the receiver's public key was used by the sender to encrypt the signed message in order to maintain confidentiality. All the best

**Complete Program Listings**

Complete listings of the programs discussed in this lesson are provided in Listing 26 and Listing 27 below.

|  |
| --- |
| /\*File Rsa09.java  Copyright 2013, Toyin  IMPORTANT: BECAUSE THIS PROGRAM USES  PREDETERMINED KEYS THAT WERE DESIGNED FOR A BLOCK  SIZE OF 12, IT WILL WORK CORRECTLY ONLY FOR A  BLOCK SIZE OF 12.  The purpose of this program is to illustrate the  signing of an unencrypted message along with the  later verification of the signature by the  receiver of the message.  The comments in this program reflect the  scenario of Alice and Bob, which is common in  discussions of RSA.  Alice needs to send a signed unencrypted message  to Bob. The message needs to be signed so that  Bob can be confident that the message that he  received was sent by Alice and not by someone  pretending to be Alice. Bob also needs to  confirm the integrity of the message by  confirming that the message was not modified  during the transmission from Alice to Bob.  Alice creates a digital signature by getting an  SHA-1 message digest for her message. Then she  encodes and encrypts the digest using her private  key and the RSA algorithm. The encrypted digest  becomes her digital signature. She signs the  message by appending the encrypted digest to the  end of the message. Bob can later decrypt the  digital signature using Alice's public key.  Bob will need some way to separate the message  from the digital signature. Before sending the  signed message to Bob, Alice appends four  characters to the end that specify the length of  the original message in characters.  Then Alice sends the signed message to Bob.  Bob extracts the message length from the end of  the signed message. He uses this information to  separate the original message from the digital  signature. Then he decrypts and decodes the  digital signature using Alice's public key.  Bob knows that the original message digest was 40  hex characters in length, because an SHA-1  message digest is always 160 bits or 20 bytes  in length. Two hex characters are required to  represent each byte, resulting in 40 hex  characters for a hex representation of an SHA-1  message digest.  Bob also knows that any additional hex characters  beyond the first 40 were the result of extending  the message digest to make it possible to  encrypt the message digest. Bob discards all but  the first 40 characters from the decoded digital  signature.  Then Bob computes an SHA-1 message digest for the  extracted message and compares it with the first  40 characters of the decoded digital signature.  If they match, he concludes that the digital  signature is valid. If not, he concludes that  the digital signature is invalid.  If the digital signature is valid, he has  confirmed the authenticity and the integrity of  the message from Alice. It could only have been  sent by Alice or by someone having access to her  private key. Furthermore, the message could not  have been modified after Alice computed her  message digest and created her digital signature.  This program supports all of the ASCII characters  from space (32) through ~ (126) inclusive to be  used in the message.  This program uses predetermined values for the  following:  Alice's public key, e  Alice's private key, d  Alice's modulus operand, n  Tested using SDK or JDK 1.5 (BlueJ as java editor) and Win7. This program produces the following output  1. Alice's keys:  2. e: 17  3. d: 279263220413  4. n: 951386374109  5. Block size: 12  6. Alice's msg: Hello Bob, how are you? I am goi  ng to go to the movie tonight. I plan to see Gone  with the Wind. I really like Clark Gable. Woul  d you like to go with me? I would really like to  have some company. Your friend, Alice.  7. Alice's msg digest: 89CA87110E2063FAF4A364B77C  18C3260AD47B8E  8. Alice's extended msg digest: 89CA87110E2063FAF  4A364B77C18C3260AD47B8E00000000  9. Alice's encrypted digital signature: 511335051  5586806264156901277970003516182249744328267808527  91138100851742238493708680633911102408  10. Alice's signed msg: Hello Bob, how are you?  I am going to go to the movie tonight. I plan to  see Gone with the Wind. I really like Clark Gabl  e. Would you like to go with me? I would really  like to have some company. Your friend, Alice.5  1133505155868062641569012779700035161822497443282  6780852791138100851742238493708680633911102408  11. Alice's signed msg with msg length appended:  Hello Bob, how are you? I am going to go to the  movie tonight. I plan to see Gone with the Wind.  I really like Clark Gable. Would you like to go  with me? I would really like to have some company. Your friend, Alice.51133505155868062641569012  7797000351618224974432826780852791138100851742238  4937086806339111024080220  12. Bob's calculated msg len: 220  13. Bob's extracted msg text: Hello Bob, how are  you? I am going to go to the movie tonight. I pl  an to see Gone with the Wind. I really like Clar  k Gable. Would you like to go with me? I would  really like to have some company. Your friend, A  lice.  14. Bob's extracted extended digital signature: 5  1133505155868062641569012779700035161822497443282  6780852791138100851742238493708680633911102408  15. Bob's decoded extended digital signature: 89C  A87110E2063FAF4A364B77C18C3260AD47B8E00000000  16. Bob's decoded digital signature: 89CA87110E20  63FAF4A364B77C18C3260AD47B8E  17. Bob's digest: 89CA87110E2063FAF4A364B77C18C32  60AD47B8E  18. Bob's conclusion: Valid signature  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  import java.math.BigInteger;  import java.security.\*;  class Rsa09{  //This is a static inner class whose sole  // purpose is to create an object that serves  // as a container for keys. It was made static  // so that it can be instantiated from within  // main. It contains precomputed values for  // the two keys, e and d, and the modulus  // operand n.  //  // e is the public key  // d is the private key  static class Keys{  //Note, these key values were designed to  // be used with a block size of 12  BigInteger n =  new BigInteger("951386374109");  BigInteger d =  new BigInteger("279263220413");  BigInteger e = new BigInteger("17");  }//end inner class Keys  public static void main(String[] args){  //Instantiate an object containing  // keys.  Keys aliceKeys = new Keys();  //Test msg.  String aliceMsg="Hello Bob, how are you? I "  + "am going to go to the movie tonight. I "  + "plan to see Gone with the Wind. I "  + "really like Clark Gable. Would you "  + "like to go with me? I would really "  + "like to have some company. Your "  + "friend, Alice.";  //Set blockSize.  int blockSize = 12;  //Instantiate an object of this class  Rsa09 obj = new Rsa09();  //Display the key values along with the  // modulus operand.  System.out.println("1. Alice's keys:");  System.out.println("2. e: " + aliceKeys.e);  System.out.println("3. d: " + aliceKeys.d);  System.out.println("4. n: " + aliceKeys.n);  System.out.println("\n5. Block size: "  + blockSize);    //Display the message that Alice will sign  // and send to Bob.  System.out.println("\n6. Alice's msg: "  + aliceMsg);    //Get the msg digest  byte[] aliceDigest =  obj.digestIt(aliceMsg.getBytes());    //Display the msg digest as a string of hex  // characters  System.out.println(  "\n7. Alice's msg digest: "  + obj.byteArrayToHexStr(aliceDigest));  //Extend the length of the digest to force it  // to be a multiple of the block size.  int tailLen = aliceDigest.length % blockSize;  int extendLen = 0;  if((tailLen > 0)){  extendLen = blockSize - tailLen;  }//end if    //Create an array for the extended digest  // with all elements initialized to a value  // of 0.  byte[] aliceExtendedDigest =  new byte[aliceDigest.length + extendLen];  //Copy the digest into the bottom of the new  // array. The empty portion at the top of  // the array becomes the extension.  System.arraycopy(aliceDigest,0,  aliceExtendedDigest,0,  aliceDigest.length);  //Convert the extended digest into a String  // of hex characters. This is necessary to  // ensure that all of the bytes are  // compatible with the characters supported  // by the encode method.  String aliceExtendedDigestAsHex =  obj.byteArrayToHexStr(aliceExtendedDigest);  //Display the extended msg digest  System.out.println(  "\n8. Alice's extended msg digest: "  + aliceExtendedDigestAsHex);  //Alice encodes the extended msg digest into  // numeric formatin preparation for  // encryption.  String aliceEncodedDigest =  obj.encode(aliceExtendedDigestAsHex);  //Alice creates a digital signature by  // encrypting the encoded msg digest using  // her private key.  String aliceSignature = obj.doRSA(  aliceEncodedDigest,aliceKeys.d,aliceKeys.n,  blockSize);  System.out.println(  "\n9. Alice's encrypted digital signature: "  + aliceSignature);  //Alice signs the msg by appending the  // digital signature onto the msg.  String aliceSignedMsg =  aliceMsg + aliceSignature;  System.out.println(  "\n10. Alice's signed msg: "  + aliceSignedMsg);    //Alice appends four characters onto the end  // of the signed msg indicating the length of  // the original msg with leading zeros if  // necessary. Bob will need this information  // to separate the msg from the signature.  // This approach can accommodate msg lengths  // up to 9999 characters.  String aliceMsgLenAsStr =  "" + aliceMsg.length();  //Confirm number of characters.  if((aliceMsgLenAsStr.length() > 4)  || (aliceMsgLenAsStr.length() <= 0)){  System.out.println(  "Message length error.");  System.exit(0);  }//end if  //Prepend leading zeros if necessary  if(aliceMsgLenAsStr.length() == 1){  aliceMsgLenAsStr = "000"  + aliceMsgLenAsStr;  }else if(aliceMsgLenAsStr.length() == 2){  aliceMsgLenAsStr = "00" + aliceMsgLenAsStr;  }else if(aliceMsgLenAsStr.length() == 3){  aliceMsgLenAsStr = "0" + aliceMsgLenAsStr;  }else if(aliceMsgLenAsStr.length() == 1){  aliceMsgLenAsStr = "000"  + aliceMsgLenAsStr;  }//end else    //Append the message length string to the  // previously signed message.  aliceSignedMsg = aliceSignedMsg  + aliceMsgLenAsStr;  System.out.println(  "\n11. Alice's signed msg with msg length "  + "appended: "  + aliceSignedMsg);    //Alice sends the signed msg to Bob.    String bobSignedMsg = aliceSignedMsg;    //At the receiving end, Bob extracts the msg  // length, the msg text, and the digital  // signature from the signed msg.  //Get the message length.  int bobMsgLen = Integer.parseInt(  bobSignedMsg.substring(  bobSignedMsg.length() - 4));  System.out.println(  "\n12. Bob's calculated msg len: "  + bobMsgLen);    //Get the message text.  String bobMsgText = bobSignedMsg.substring(  0,bobMsgLen);  System.out.println(  "\n13. Bob's extracted msg text: "  + bobMsgText);    //Bob knows that everything following the msg  // text except for the four characters at the  // end that indicate the message length is  // the encoded and encrypted version of the  // extended digital signature. He extracts  // it.  String bobExtractedSignature =  bobSignedMsg.substring(  bobMsgLen,bobSignedMsg.length() - 4);  System.out.println(  "\n14. Bob's extracted extended digital "  + "signature: "  + bobExtractedSignature);  //Now Bob has the message length, the  // message, and the encrypted digital  // signature.  //Bob decrypts the extended digital  // signature using Alice's public key.  String bobDecryptedExtendedSignature =  obj.doRSA(bobExtractedSignature,  aliceKeys.e,aliceKeys.n,blockSize);  //Bob decodes the extended digital signature.  String bobDecodedExtendedSignature =  obj.decode(  bobDecryptedExtendedSignature);  System.out.println(  "\n15. Bob's decoded extended digital "  + "signature: "  + bobDecodedExtendedSignature);    //Bob knows that the digital signature  // consists of 40 hex digits and that any  // additional characters were put there to  // fill out the last block and make  // encryption possible. He extracts the hex  // version of the digital signature by  // ignoring all but the first 40 hex  // characters.  String bobDecodedSignature =  bobDecodedExtendedSignature.  substring(0,40);  System.out.println(  "\n16. Bob's decoded digital signature: "  + bobDecodedSignature);    //Bob computes the msg digest for comparison  // with the decoded signature.  byte[] bobDigest =  obj.digestIt(bobMsgText.getBytes());    System.out.println("\n17. Bob's digest: "  + obj.byteArrayToHexStr(bobDigest));    //Bob compares the decoded digital signature  // with his computed msg digest and displays  // an output message indicating whether or  // not the signature is valid.  if(bobDecodedSignature.equals(  obj.byteArrayToHexStr(bobDigest))){  System.out.println(  "\n18. Bob's conclusion: "  + "Valid signature");  }else{  System.out.println(  "\n18. Bob's conclusion: "  + "Invalid signature");  }//end else  }//end main  //-------------------------------------------//  //The purpose of this method is to encode a  // plain text msg into numeric format  // where:  // space = 32 - 32 = 0  // A = 65 - 32 = 33  // ...  // Z = 90 - 32 = 58  // ...  // a = 97 - 32 = 65  // ...  // ~ = 126 - 32 = 94  //Note that this encoding method supports all  //of the ASCII characters from space through  // tilde (~) inclusive.  String encode(String msg){  byte[] textChars = msg.getBytes();  String temp = "";  String encodedMsg = "";  //Build the encoded text string two numeric  // characters at a time. Each msg  // character is converted into two numeric  // characters according to the relationships  // given above.  for(int cnt = 0; cnt < msg.length();  cnt++){  temp = String.valueOf(  textChars[cnt] - ' ');  //Convert all single-character numeric  // values to two characters with a leading  // zero, as in 09.  if(temp.length() < 2) temp = "0" + temp;  encodedMsg += temp;  }//end for loop  return encodedMsg;  }//end encode  //-------------------------------------------//  //The purpose of this method is to reverse the  // encoding process implemented by the encode  // method, converting a string of numeric  // characters back to a text string containing  // the ASCII characters from space through  // tilde.  String decode(String encodedMsg){  String temp = "";  String decodedText = "";  for(int cnt = 0; cnt < encodedMsg.length();  cnt += 2){  temp = encodedMsg.substring(cnt,cnt + 2);  //Convert two numeric text characters to a  // value of type int.  int val = Integer.parseInt(temp) + 32;  //Convert the ASCII character values to  // numeric String values and build the  // output String one character at a time.  decodedText += String.valueOf((char)val);  }//end for loop  return decodedText;  }//end decode  //-------------------------------------------//  //Apply the RSA algorithm to an input string  // using the exponent exp and the modulus  // operator n, which are provided as input  // parameters. This method can be used to  // encrypt or to decipher the input string  // depending on whether the exponent is an  // encryption key or a decryption key. Apply  // the algorithm for the block size given by  // the incoming parameter named blockSize.  String doRSA(String inputString,  BigInteger exp,  BigInteger n,  int blockSize){  BigInteger block;  BigInteger output;  String temp = "";  String outputString = "";  //Iterate and process one block at a time.  for(int cnt = 0; cnt < inputString.length();  cnt += blockSize){  //Get the next block of characters  // and encapsulate them in a BigInteger  // object.  temp = inputString.substring(  cnt,cnt + blockSize);  block = new BigInteger(temp);  //Raise the block to the power exp, apply  // the modulus operand n, and save the  // remainder. This is the essence of the  // RSA algorithm.  output = block.modPow(exp,n);  //Convert the numeric result to a  // four-character string, appending leading  // zeros as necessary.  temp = output.toString();  while(temp.length() < blockSize){  temp = "0" + temp;  }//end while  //Build the outputString blockSize  // characters at a time. Each character  // in the inputString results in one  // character in the outputString.  outputString += temp;  }//end for loop  return outputString;  }//end doRSA  //-------------------------------------------//    //This method generates and returns a digest  // for an incoming array of bytes using Sun's  // SHA msg digest algorithm..  byte[] digestIt(byte[] dataIn){  byte[] theDigest = null;  try{  //Create a MessageDigest object  // implementing the SHA algorithm, as  // supplied by SUN  MessageDigest msgDigest =  MessageDigest.getInstance("SHA", "SUN");  //Feed the byte array to the digester. Can  // accommodate multiple calls if needed  msgDigest.update(dataIn);  //Complete the digestion and save the  // result  theDigest = msgDigest.digest();  }catch(Exception e){System.out.println(e);}  //Return the digest value to the calling  // method as an array of bytes.  return theDigest;  }//end digestIt()  //-------------------------------------------//    //This method converts an incoming array of  // bytes into a string that represents each of  // the bytes as two hex characters.  String byteArrayToHexStr(byte[] data){  String output = "";  String tempStr = "";  int tempInt = 0;  for(int cnt = 0;cnt < data.length;cnt++){  //Deposit a byte into the 8 lsb of an int.  tempInt = data[cnt]&0xFF;  //Get hex representation of the int as a  // string.  tempStr = Integer.toHexString(tempInt);  //Append a leading 0 if necessary so that  // each hex string will contain two  // characters.  if(tempStr.length() == 1)  tempStr = "0" + tempStr;  //Concatenate the two characters to the  // output string.  output = output + tempStr;  }//end for loop  return output.toUpperCase();  }//end byteArrayToHexStr  //-------------------------------------------//  }//end class Rsa09  **Listing 26** |

|  |
| --- |
| /\*File Rsa10.java  Copyright 2013, Toyin  IMPORTANT: BECAUSE THIS PROGRAM USES  PREDETERMINED KEYS THAT WERE DESIGNED FOR A BLOCK  SIZE OF 12, IT WILL WORK CORRECTLY ONLY FOR A  BLOCK SIZE OF 12.  The purpose of this program is to illustrate the  signing of an encrypted message along with the  later verification of the signature by the  receiver of the message.  The comments in this program reflect the  scenario of Alice and Bob, which is common in  discussions of RSA.  Alice needs to send a signed encrypted message  to Bob. The message needs to be signed so that  Bob can be confident that the message that he  received was sent by Alice and not by someone  pretending to be Alice. Bob also needs to  confirm the integrity of the message by  confirming that the message was not modified  during the transmission from Alice to Bob.  Finally, the confidentiality of the message must  be maintained. That is, the message must be sent  in such a way that it can only be read by Bob.  Alice creates a digital signature by getting an  SHA-1 message digest for her message. Then she  encodes and encrypts the digest using her private  key and the RSA algorithm. The encrypted digest  becomes her digital signature. She signs the  message by appending the encrypted digest to the  end of the message. Bob can later decrypt the  digital signature using Alice's public key.  Alice will ensure confidentiality by encrypting  the entire signed message using Bob's public key.  Then only Bob or someone having access to his  private key can decrypt and read the message.  Bob will need some way to separate the message  from the digital signature after he decrypts and  decodes it. One way to make this possible is to  notify Bob of the length of the original message.  In addition, before encoding and encrypting the  signed message, Alice must extend the length of  the signed message to cause the length of the  signed and encoded message to be a multiple of  the block size.  Alice takes care of both of these requirement at  the same time. She pads the unencoded signed  message with a pad that causes the extended  length of the unencoded signed message to be a  multiple of half the block size (the encoding  scheme in use doubles the length of the signed  message). In addition, she causes the last four  bytes in the pad to contain numeric characters  specifying the length of the original message.  If the pad is more than four bytes long, Alice  fills in the pad with equal characters (=).  Bob can later extract the length of the  original message and use it to separate the  message from the signature.  Then Alice sends the signed encrypted message  to Bob.  Bob begins by decrypting and decoding the  encrypted signed message. He extracts the  length of the original message from the end of  the decoded signed message. He uses this  information along with his knowledge that the  pad may also contain equal characters (=) as  filler to extract the encrypted digital signature  from the decoded signed message. He also uses  this information to extract the message from the  decoded signed message.  Bob decrypts and decodes the encrypted digital  signature using Alice's public key.  Bob knows that the original message digest was 40  hex characters in length, because an SHA-1  message digest is always 160 bits or 20 bytes  in length. Two hex characters are required to  represent each byte, resulting in 40 hex  characters for a hex representation of an SHA-1  message digest.  Bob also knows that any additional hex characters  beyond the first 40 were the result of extending  the message digest to make it possible to  encrypt it. Bob discards all but the first 40  characters from the decoded digital signature.  Then Bob computes an SHA-1 message digest for the  extracted message and compares it with the first  40 characters of the decoded digital signature.  If they match, he concludes that the digital  signature is valid. If not, he concludes that  the digital signature is invalid.  If the digital signature is valid, he has  confirmed the authenticity and the integrity of  the message from Alice. It could only have been  sent by Alice or by someone having access to her  private key. Furthermore, the message could not  have been modified after Alice computed her  message digest and created her digital signature.  Finally, because Alice encrypted the signed  message using Bob's public key, only Bob or  someone having access to his private key can  decrypt the signed message. Therefore, the  confidentiality of the message was protected  during the transmission of the message from Alice  to Bob.  This program supports all of the ASCII characters  from space (32) through ~ (126) inclusive to be  used in the message. None of these characters  are set aside to be used as delimiter characters.  The use of the equal character (=) to extend the  signed message prior to encoding and encryption  does not prohibit the use of this character  within the message proper.  This program uses predetermined values for the  following:  Alice's public key, e  Alice's private key, d  Alice's modulus operand, n  Bob's public key, e  Bob's private key, d  Bob's modulus operand, n  The above keys were all designed to be used with  a block size of 12.  Tested using SDK or JDK 1.5(BlueJ as java editor) and Win7. This program produces the following output  1. Alice's keys:  2. e: 17  3. d: 279263220413  4. n: 951386374109  5. Bob's keys:  6. e: 19  7. d: 799574694235  8. n: 951386374109  9. Block size: 12  10. Alice's msg: Hello Bob, how are you? I am go  ing to go to the movie tonight. I plan to see Gon  e with the Wind. I really like Clark Gable. Wou  ld you like to go with me? I would really like t  o have some company. Your friend, Alice.  11. Alice's msg digest: 89CA87110E2063FAF4A364B77  C18C3260AD47B8E  12. Alice's extended msg digest: 89CA87110E2063FA  F4A364B77C18C3260AD47B8E00000000  13. Alice's encrypted digital signature: 51133505  1558680626415690127797000351618224974432826780852  791138100851742238493708680633911102408  14. Alice's signed msg: Hello Bob, how are you?  I am going to go to the movie tonight. I plan to  see Gone with the Wind. I really like Clark Gabl  e. Would you like to go with me? I would really  like to have some company. Your friend, Alice.5  1133505155868062641569012779700035161822497443282  6780852791138100851742238493708680633911102408  15. Alice's padded signed msg with msg length app  ended: Hello Bob, how are you? I am going to go t  o the movie tonight. I plan to see Gone with the  Wind. I really like Clark Gable. Would you like  to go with me? I would really like to have some  company. Your friend, Alice.5113350515586806264  1569012779700035161822497443282678085279113810085  1742238493708680633911102408====0220  16. Alice's encrypted signed message: 91401046409  0477812185023522059255227103733444908183675749115  8834555666464747955317487974740038223011505609617  0958348608789466477424942329033371476785906742456  0777331120891074470078287368038504581398329728289  1169705147103312551664808525983850773644838883849  5892974479228714690946361944896047795453704351458  0138918238467231200275709874384289825441638463619  4489607764755499173131264107586650093298351648729  8983371264328395673083207830600854525558017959215  2887426943639881827352686383004770007421461800123  9893565435323832631156747443170942361455453339508  2810225070374501073338462893929842434027331818342  2870733493040858128661542384978031710885942868854  17. Bob's decrypted and decoded signed message: H  ello Bob, how are you? I am going to go to the m  ovie tonight. I plan to see Gone with the Wind.  I really like Clark Gable. Would you like to go  with me? I would really like to have some compan  y. Your friend, Alice.51133505155868062641569012  7797000351618224974432826780852791138100851742238  493708680633911102408====0220  18. Bob's calculated msg len: 220  19. Bob's extracted msg text: Hello Bob, how are  you? I am going to go to the movie tonight. I pl  an to see Gone with the Wind. I really like Clar  k Gable. Would you like to go with me? I would  really like to have some company. Your friend, A  lice.  20. Bob's extracted extended digital signature: 5  1133505155868062641569012779700035161822497443282  6780852791138100851742238493708680633911102408  21. Bob's decoded extended digital signature: 89C  A87110E2063FAF4A364B77C18C3260AD47B8E00000000  22. Bob's decoded digital signature: 89CA87110E20  63FAF4A364B77C18C3260AD47B8E  23. Bob's digest: 89CA87110E2063FAF4A364B77C18C32  60AD47B8E  24. Bob's conclusion: Valid signature  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  import java.math.BigInteger;  import java.security.\*;  class Rsa10{  //This is a static inner class whose sole  // purpose is to create an object that serves  // as a container for keys. It was made static  // so that it can be instantiated from within  // main. It contains precomputed values for  // the two keys, e and d, and the modulus  // operand n.  //  // e is the public key  // d is the private key  static class Keys{  BigInteger n;  BigInteger d;  BigInteger e;  }//end inner class Keys  public static void main(String[] args){  //Instantiate an object containing  // Alice's keys.  Keys aliceKeys = new Keys();  //Note, these key values were designed to  // be used with a block size of 12  aliceKeys.n = new BigInteger("951386374109");  aliceKeys.d = new BigInteger("279263220413");  aliceKeys.e = new BigInteger("17");    //Instantiate an object containing  // Bob's keys  Keys bobKeys = new Keys();  //Note, these key values were designed to  // be used with a block size of 12  bobKeys.n = new BigInteger("951386374109");  bobKeys.d = new BigInteger("799574694235");  bobKeys.e = new BigInteger("19");  //Test msg.  String aliceMsg="Hello Bob, how are you? I "  + "am going to go to the movie tonight. I "  + "plan to see Gone with the Wind. I "  + "really like Clark Gable. Would you "  + "like to go with me? I would really "  + "like to have some company. Your "  + "friend, Alice.";  //Set blockSize.  int blockSize = 12;  //Instantiate an object of this class  Rsa10 obj = new Rsa10();  //Display the key values along with the  // modulus operand.  System.out.println("1. Alice's keys:");  System.out.println("2. e: " + aliceKeys.e);  System.out.println("3. d: " + aliceKeys.d);  System.out.println("4. n: " + aliceKeys.n);    System.out.println("\n5. Bob's keys:");  System.out.println("6. e: " + bobKeys.e);  System.out.println("7. d: " + bobKeys.d);  System.out.println("8. n: " + bobKeys.n);    System.out.println("\n9. Block size: "  + blockSize);    //Display the message that Alice will sign  // and send to Bob.  System.out.println("\n10. Alice's msg: "  + aliceMsg);    //Get the msg digest  byte[] aliceDigest =  obj.digestIt(aliceMsg.getBytes());    //Display the msg digest as a string of hex  // characters  System.out.println(  "\n11. Alice's msg digest: "  + obj.byteArrayToHexStr(aliceDigest));  //Extend the length of the digest to force it  // to be a multiple of the block size.  int tailLen = aliceDigest.length % blockSize;  int extendLen = 0;  if((tailLen > 0)){  extendLen = blockSize - tailLen;  }//end if    //Create an array for the extended digest  // with all elements initialized to a value  // of 0.  byte[] aliceExtendedDigest =  new byte[aliceDigest.length + extendLen];  //Copy the digest into the bottom of the new  // array. The empty portion at the top of  // the array becomes the extension.  System.arraycopy(aliceDigest,0,  aliceExtendedDigest,0,  aliceDigest.length);  //Convert the extended digest into a String  // of hex characters. This is necessary to  // ensure that all of the bytes are  // compatible with the characters supported  // by the encode method.  String aliceExtendedDigestAsHex =  obj.byteArrayToHexStr(aliceExtendedDigest);  //Display the extended msg digest  System.out.println(  "\n12. Alice's extended msg digest: "  + aliceExtendedDigestAsHex);  //Alice encodes the extended msg digest into  // numeric format in preparation for  // encryption.  String aliceEncodedDigest =  obj.encode(aliceExtendedDigestAsHex);  //Alice creates a digital signature by  // encrypting the encoded msg digest using  // her private key.  String aliceSignature = obj.doRSA(  aliceEncodedDigest,aliceKeys.d,aliceKeys.n,  blockSize);  System.out.println(  "\n13. Alice's encrypted digital "  + "signature: "+ aliceSignature);  //Alice signs the msg by appending the  // digital signature onto the msg.  String aliceSignedMsg =  aliceMsg + aliceSignature;  System.out.println(  "\n14. Alice's signed msg: "  + aliceSignedMsg);    //Because she needs to maintain  // confidentiality, Alice will encrypt the  // signed message using Bob's public key  // before sending the message to him. In  // preparation for encoding and encryption,  // she needs to extend the length of the  // signed message such that the total length  // is a multiple of half the block size. In  // addition, she needs to notify Bob as to  // the length of the original message so  // that he will have a way to separate the  // text of the original message from the  // encrypted message signature. She will  // accomplish both of these requirements at  // the same time. She will pad the signed  // message using a pad that includes the  // length of the original message in the  // last four bytes of the pad. All bytes in  // the pad other than the last four bytes  // are filled with equal characters (=).  String alicePaddedSignedMsg =  obj.padTheMsg(aliceSignedMsg,blockSize/2,  aliceMsg.length());  System.out.println(  "\n15. Alice's padded signed msg with msg "  + "length appended: "  + alicePaddedSignedMsg);    //Alice encodes and encrypts the padded  // signed message using Bob's public key.    //Encode the message.  String aliceEncodedSignedMsg =  obj.encode(alicePaddedSignedMsg);  //Encrypt the message.  String aliceEncryptedSignedMsg = obj.doRSA(  aliceEncodedSignedMsg,bobKeys.e,bobKeys.n,  blockSize);  System.out.println(  "\n16. Alice's encrypted signed message: "  + aliceEncryptedSignedMsg);    //\*\*\*\*\*\*\*\*\*\*\*\*  //Alice sends the encrypted signed msg to  // Bob.  //\*\*\*\*\*\*\*\*\*\*\*\*    String bobEncryptedSignedMsg =  aliceEncryptedSignedMsg;    //At the receiving end, Bob begins by  // decrypting and decoding the signed  // message using his private key.  //Decrypt the message.  String bobDecryptedSignedMsg = obj.doRSA(  bobEncryptedSignedMsg,bobKeys.d,bobKeys.n,  blockSize);  //Decode the message.  String bobDecodedSignedMsg =  obj.decode(bobDecryptedSignedMsg);    System.out.println(  "\n17. Bob's decrypted and decoded signed "  + "message: "+ bobDecodedSignedMsg);    //Bob extracts the message length, the  // message text, and the digital signature  // from the decoded signed msg.  //Extract the message length.  int bobMsgLen = Integer.parseInt(  bobDecodedSignedMsg.substring(  bobDecodedSignedMsg.length() - 4));  System.out.println(  "\n18. Bob's calculated msg len: "  + bobMsgLen);    //Extract the message text.  String bobExtractedMsgText =  bobDecodedSignedMsg.substring(  0,bobMsgLen);  System.out.println(  "\n19. Bob's extracted msg text: "  + bobExtractedMsgText);    //Bob knows that everything between the  // message text and the first equal sign in  // the decoded signed message is the  // extended, encoded, and encrypted digital  // signature. He also knows that if there  // are no equal signs the last four bytes  // containing the length of the original  // message need to be discarded. He uses  // this knowledge to extract the encrypted  // signature.  String bobExtractedEncryptedSignature;  if(bobDecodedSignedMsg.indexOf("=") != -1){  bobExtractedEncryptedSignature =  bobDecodedSignedMsg.substring(  bobMsgLen,bobDecodedSignedMsg.indexOf(  "="));  }else{  bobExtractedEncryptedSignature =  bobDecodedSignedMsg.substring(  bobMsgLen,  bobDecodedSignedMsg.length() - 4);  }//end else    System.out.println(  "\n20. Bob's extracted extended digital "  + "signature: "  + bobExtractedEncryptedSignature);  //Now Bob has the message length, the  // message, and the encrypted digital  // signature.  //Bob decrypts the extended digital  // signature using Alice's public key.  String bobDecryptedExtendedSignature =  obj.doRSA(bobExtractedEncryptedSignature,  aliceKeys.e,aliceKeys.n,blockSize);  //Bob decodes the extended digital signature.  String bobDecodedExtendedSignature =  obj.decode(bobDecryptedExtendedSignature);  System.out.println(  "\n21. Bob's decoded extended digital "  + "signature: "  + bobDecodedExtendedSignature);    //Bob knows that the digital signature  // consists of only 40 hex digits and that  // any additional characters were put there  // to fill out the last block and make  // encryption possible. He extracts the hex  // version of the digital signature by  // ignoring all but the first 40 hex  // characters.  String bobDecodedSignature =  bobDecodedExtendedSignature.substring(  0,40);  System.out.println(  "\n22. Bob's decoded digital signature: "  + bobDecodedSignature);    //Bob computes the msg digest for comparison  // with the decoded signature.  byte[] bobDigest = obj.digestIt(  bobExtractedMsgText.getBytes());    System.out.println("\n23. Bob's digest: "  + obj.byteArrayToHexStr(bobDigest));    //Bob compares the decoded digital signature  // with his computed msg digest and displays  // an output message indicating whether or  // not the signature is valid.  if(bobDecodedSignature.equals(  obj.byteArrayToHexStr(bobDigest))){  System.out.println(  "\n24. Bob's conclusion: "  + "Valid signature");  }else{  System.out.println(  "\n24. Bob's conclusion: "  + "Invalid signature");  }//end else  }//end main  //-------------------------------------------//  //The purpose of this method is to encode a  // plain text msg into numeric format  // where:  // space = 32 - 32 = 0  // A = 65 - 32 = 33  // ...  // Z = 90 - 32 = 58  // ...  // a = 97 - 32 = 65  // ...  // ~ = 126 - 32 = 94  //Note that this encoding method supports all  //of the ASCII characters from space through  // tilde (~) inclusive.  String encode(String msg){  byte[] textChars = msg.getBytes();  String temp = "";  String encodedMsg = "";  //Build the encoded text string two numeric  // characters at a time. Each msg  // character is converted into two numeric  // characters according to the relationships  // given above.  for(int cnt = 0; cnt < msg.length();  cnt++){  temp = String.valueOf(  textChars[cnt] - ' ');  //Convert all single-character numeric  // values to two characters with a leading  // zero, as in 09.  if(temp.length() < 2) temp = "0" + temp;  encodedMsg += temp;  }//end for loop  return encodedMsg;  }//end encode  //-------------------------------------------//  //The purpose of this method is to reverse the  // encoding process implemented by the encode  // method, converting a string of numeric  // characters back to a text string containing  // the ASCII characters from space through  // tilde.  String decode(String encodedMsg){  String temp = "";  String decodedText = "";  for(int cnt = 0; cnt < encodedMsg.length();  cnt += 2){  temp = encodedMsg.substring(cnt,cnt + 2);  //Convert two numeric text characters to a  // value of type int.  int val = Integer.parseInt(temp) + 32;  //Convert the ASCII character values to  // numeric String values and build the  // output String one character at a time.  decodedText += String.valueOf((char)val);  }//end for loop  return decodedText;  }//end decode  //-------------------------------------------//  //Apply the RSA algorithm to an input string  // using the exponent exp and the modulus  // operator n, which are provided as input  // parameters. This method can be used to  // encrypt or to decipher the input string  // depending on whether the exponent is an  // encryption key or a decryption key. Apply  // the algorithm for the block size given by  // the incoming parameter named blockSize.  String doRSA(String inputString,  BigInteger exp,  BigInteger n,  int blockSize){  BigInteger block;  BigInteger output;  String temp = "";  String outputString = "";  //Iterate and process one block at a time.  for(int cnt = 0; cnt < inputString.length();  cnt += blockSize){  //Get the next block of characters  // and encapsulate them in a BigInteger  // object.  temp = inputString.substring(  cnt,cnt + blockSize);  block = new BigInteger(temp);  //Raise the block to the power exp, apply  // the modulus operand n, and save the  // remainder. This is the essence of the  // RSA algorithm.  output = block.modPow(exp,n);  //Convert the numeric result to a  // four-character string, appending leading  // zeros as necessary.  temp = output.toString();  while(temp.length() < blockSize){  temp = "0" + temp;  }//end while  //Build the outputString blockSize  // characters at a time. Each character  // in the inputString results in one  // character in the outputString.  outputString += temp;  }//end for loop  return outputString;  }//end doRSA  //-------------------------------------------//    //This method generates and returns a digest  // for an incoming array of bytes using Sun's  // SHA msg digest algorithm..  byte[] digestIt(byte[] dataIn){  byte[] theDigest = null;  try{  //Create a MessageDigest object  // implementing the SHA algorithm, as  // supplied by SUN  MessageDigest msgDigest =  MessageDigest.getInstance("SHA", "SUN");  //Feed the byte array to the digester. Can  // accommodate multiple calls if needed  msgDigest.update(dataIn);  //Complete the digestion and save the  // result  theDigest = msgDigest.digest();  }catch(Exception e){System.out.println(e);}  //Return the digest value to the calling  // method as an array of bytes.  return theDigest;  }//end digestIt()  //-------------------------------------------//    //This method converts an incoming array of  // bytes into a string that represents each of  // the bytes as two hex characters.  String byteArrayToHexStr(byte[] data){  String output = "";  String tempStr = "";  int tempInt = 0;  for(int cnt = 0;cnt < data.length;cnt++){  //Deposit a byte into the 8 lsb of an int.  tempInt = data[cnt]&0xFF;  //Get hex representation of the int as a  // string.  tempStr = Integer.toHexString(tempInt);  //Append a leading 0 if necessary so that  // each hex string will contain two  // characters.  if(tempStr.length() == 1)  tempStr = "0" + tempStr;  //Concatenate the two characters to the  // output string.  output = output + tempStr;  }//end for loop  return output.toUpperCase();  }//end byteArrayToHexStr  //-------------------------------------------//  //This method pads a message such that the  // final length is a multiple of block and the  // last four bytes specify the origMsgLength.  // All the remaining bytes in the pad are  // filled with equal characters (=). The padded  // message is returned as type String.  private String padTheMsg(  String msgIn,int block,int origMsgLen){  byte[] msgData = msgIn.getBytes();  int msgInLen = msgData.length;  int tailLength = msgInLen%block;  int padLength = 0;  if((block - tailLength >= 4))  padLength = block - tailLength;  else  padLength = 2\*block - tailLength;    //Create a four-byte array containing  // characters that indicate the length of  // the original msg with leading zeros if  // necessary.  String msgLenAsStr = "" + origMsgLen;  //Confirm number of characters.  if((msgLenAsStr.length() > 4)  || (msgLenAsStr.length() <= 0)){  System.out.println(  "Message length error.");  System.exit(0);  }//end if    //Prepend leading zeros if necessary  if(msgLenAsStr.length() == 1){  msgLenAsStr = "000" + msgLenAsStr;  }else if(msgLenAsStr.length() == 2){  msgLenAsStr = "00" + msgLenAsStr;  }else if(msgLenAsStr.length() == 3){  msgLenAsStr = "0" + msgLenAsStr;  }else if(msgLenAsStr.length() == 1){  msgLenAsStr = "000" + msgLenAsStr;  }//end else    //Create a four-byte array containing the  // original message length as four numeric  // characters with leading zeros.  byte[] msgLenAsBytes =  msgLenAsStr.getBytes();    //Construct an array containing the bytes  // required to make the padded message length  // equal to a multiple of block  byte[] thePad = new byte[padLength];  //Populate the array with = characters.  // No need to put = characters in the last  // four bytes.  for(int cnt = 0;cnt < thePad.length - 4;  cnt++){  thePad[cnt] = '=';  }//end for loop    //Put the original message length at the end  // of the pad.  System.arraycopy(msgLenAsBytes,0,thePad,  thePad.length - 4,4);    //Create an output array.  byte[] output =  new byte[msgInLen + padLength];  //Populate the output array with the original  // msgData concatenated with the pad.  System.arraycopy(msgData,0,output,0,  msgInLen);  System.arraycopy(  thePad,0,output,msgInLen,thePad.length);    //Convert the output to a String and return  // the String..  String outputAsStr = new String(output);  return outputAsStr;  }//end padTheMsg  //-------------------------------------------//  }//end class Rsa10  **Listing 27** |

Thats all for the day.