Appendix A – Initial Ledger and Blockchain for Example in Paper

```
Ledger before transaction:
Public_Key:
30819f300d06092a864886f70d010101050003818d00308189028181009f1de4c291970420e8728c0e23a9
737c20d2b68ab91edf9c917e586b851abcca902f5a55db8a713fb2de6fbd63fe0269eea667af0211bf01d3c
04c30e193de96d7d77b32c294c5dfd376ca86dff651881a0de8f32aed2ad486c855ab727d0e185b83a798c1
8bfd558e11e83c7c9f6b07383d5e7d2594d5751a9b75df9eff4590203010001
Unused_Transactions:
de5f04609e97985626e9b8ab294272cd276dcdf83031b4b7b9eb2754042ac007 10
2d0cce04e54d257a157f16b0d42c77e1076ebd114c49479b1163e24ae54ea5e4 88020.0
Public_Key:
30819f300d06092a864886f70d010101050003818d00308189028181009bda7757642474dc726b914db3cb0
98f30e36fb9dcd383437b1d776b871bb741d5fca7f762b9d7715ba91e3553302a83821c5302e783fc3bbf09
2cbc65195426c6004dfc472395bb376a6500bf609d7ca7256dcbd6704991407034b010f27192c3f610e924c
461316fed6f1c8669520df00f6dd86cda28970be3bf330d416ed70203010001
Unused_Transactions:
974dadd9665237a50a386904e38804e79842d8969aac563516edeb618bcad8ba 5000.0
2d0cce04e54d257a157f16b0d42c77e1076ebd114c49479b1163e24ae54ea5e4 10
Public_Key:
30819f300d06092a864886f70d010101050003818d00308189028181009b0d59dcddfd2d4e2b879251a4bc24
eb5542e8258ecb310c79c009e752429732f5f3e76d72f8f2f4d122fa1b090109637b9f99ea71fe105f376eb
f0eba0f2ed66dd12919103c54e62bc3677b3c148c9670f20fe26d40c04535db0c19568e062d81d7f9e37c76
53c8e75a10d5f93003dcbb0cbade46c23d77f7a861cf3e0a51ebb0203010001
Unused_Transactions:
de5f04609e97985626e9b8ab294272cd276dcdf83031b4b7b9eb2754042ac007 4000.0
2d0cce04e54d257a157f16b0d42c77e1076ebd114c49479b1163e24ae54ea5e4 3000.0

Blockchain before transaction:
New Block:

Previous_Block_Hash:
firstblock

To:
30819f300d06092a864886f70d010101050003818d00308189028181009f1de4c291970420e8728c0e23a9
737c20d2b68ab91edf9c917e586b851abcca902f5a55db8a713fb2de6fbd63fe0269eea667af0211bf01d3c
04c30e193de96d7d77b32c294c5dfd376ca86dff651881a0de8f32aed2ad486c855ab727d0e185b83a798c1
8bfd558e11e83c7c9f6b07383d5e7d2594d5751a9b75df9eff4590203010001

From:
30819f300d06092a864886f70d010101050003818d00308189028181009f1de4c291970420e8728c0e23a9
737c20d2b68ab91edf9c917e586b851abcca902f5a55db8a713fb2de6fbd63fe0269eea667af0211bf01d3c
04c30e193de96d7d77b32c294c5dfd376ca86dff651881a0de8f32aed2ad486c855ab727d0e185b83a798c1
8bfd558e11e83c7c9f6b07383d5e7d2594d5751a9b75df9eff4590203010001

Amount:
100000

Hash:
c1c88c8bdcd4127b0e29b77d6948d5f332624df9f785c5ea0f957ed515b4c99e

Signature:
[B@58b8dfdf

Inputs:
firsttransaction 100000

```

```
Outputs:
c1c88c8bdcd4127b0e29b77d6948d5f332624df9f785c5ea0f957ed515b4c99e 0.0
c1c88c8bdcd4127b0e29b77d6948d5f332624df9f785c5ea0f957ed515b4c99e 100000.0

Miner_Reward_Public_Key:
30819f300d06092a864886f70d010101050003818d00308189028181009f1de4c291970420e8728c0e23a9
737c20d2b68ab91edf9c917e586b851abcca902f5a55db8a713fb2de6fbd63fe0269eea667af0211bf01d3c
04c30e193de96d7d77b32c294c5dfd376ca86dff651881a0de8f32aed2ad486c855ab727d0e185b83a798c1
8bfd558e11e83c7c9f6b07383d5e7d2594d5751a9b75df9eff4590203010001

Miner_Solution:
9>j

Block_Hash:
0000a2a9dd42bcf8e2d975ceb621c38620a27d7f7d4b38eeb2bc0577b5645b26

New Block:

Previous_Block_Hash:
0000a2a9dd42bcf8e2d975ceb621c38620a27d7f7d4b38eeb2bc0577b5645b26

To:
30819f300d06092a864886f70d010101050003818d00308189028181009bda7757642474dc726b914db3cb0
98f30e36fb9dcd383437b1d776b871bb741d5fca7f762b9d7715ba91e3553302a83821c5302e783fc3bbf09
2cbc65195426c6004dfc472395bb376a6500bf609d7ca7256dcbd6704991407034b010f27192c3f610e924c
461316fed6f1c8669520df00f6dd86cda28970be3bf330d416ed70203010001

From:
30819f300d06092a864886f70d010101050003818d00308189028181009f1de4c291970420e8728c0e23a9
737c20d2b68ab91edf9c917e586b851abcca902f5a55db8a713fb2de6fbd63fe0269eea667af0211bf01d3c
04c30e193de96d7d77b32c294c5dfd376ca86dff651881a0de8f32aed2ad486c855ab727d0e185b83a798c1
8bfd558e11e83c7c9f6b07383d5e7d2594d5751a9b75df9eff4590203010001

Amount:
5000

Hash:
974dadd9665237a50a386904e38804e79842d8969aac563516edeb618bcad8ba

Signature:
[B@5171d6fa

Inputs:
c1c88c8bdcd4127b0e29b77d6948d5f332624df9f785c5ea0f957ed515b4c99e 100000.0

Outputs:
974dadd9665237a50a386904e38804e79842d8969aac563516edeb618bcad8ba 95000.0
974dadd9665237a50a386904e38804e79842d8969aac563516edeb618bcad8ba 5000.0

Miner_Reward_Public_Key:
30819f300d06092a864886f70d010101050003818d00308189028181009f1de4c291970420e8728c0e23a9
737c20d2b68ab91edf9c917e586b851abcca902f5a55db8a713fb2de6fbd63fe0269eea667af0211bf01d3c
04c30e193de96d7d77b32c294c5dfd376ca86dff651881a0de8f32aed2ad486c855ab727d0e185b83a798c1
8bfd558e11e83c7c9f6b07383d5e7d2594d5751a9b75df9eff4590203010001

Miner_Solution:
C>a

Block_Hash:
000086e478e73f01b47165b2356ad5ba7414933b277e76ba9b45340c803fc6e5

```

```
New Block:

Previous_Block_Hash:
000086e478e73f01b47165b2356ad5ba7414933b277e76ba9b45340c803fc6e5

To:
30819f300d06092a864886f70d010101050003818d00308189028181009 0b0d59dcddfd2d4e2b879251a4bc24eb5542e8258ecb310c79c009e752429732f5f3e76d72f8f2f4d122fa1b090109637b9f99ea71fe105f376ebf0eba0f2ed66dd12919103c54e62bc3677b3c148c9670f20fe26d40c04535db0c19568e062d81d7f9e37c7653c8e75a10d5f93003dcbb0cbade46c23d77f7a861cf3e0a51ebb0203010001

From:
        30819f300d06092a864886f70d010101050003818d00308189028181009 0f1de4c291970420e8728c0e23a9737c20d2b68ab91edf9c917e586b851abcca902f5a55db8a713fb2de6fbd63fe0269eea667af0211bf01d3c04c30e193de96d7d77b32c294c5dfd376ca86dff651881a0de8f32aed2ad486c855ab727d0e185b83a798c18bfd558e11e83c7c9f6b07383d5e7d2594d5751a9b75df9eff4590203010001

        Amount:
        4000

        Hash:
        de5f04609e97985626e9b8ab294272cd276dcdf83031b4b7b9eb2754042ac007

        Signature:
        [B@8d8e0dc

        Inputs:
        c1c88c8bdcd4127b0e29b77d6948d5f332624df9f785c5ea0f957ed515b4c99e 10
        974dadd9665237a50a386904e38804e79842d8969aac563516edeb618bcad8ba 95000.0

        Outputs:
        de5f04609e97985626e9b8ab294272cd276dcdf83031b4b7b9eb2754042ac007 91010.0
        de5f04609e97985626e9b8ab294272cd276dcdf83031b4b7b9eb2754042ac007 4000.0

        Miner_Reward_Public_Key:
        30819f300d06092a864886f70d010101050003818d00308189028181009 0f1de4c291970420e8728c0e23a9737c20d2b68ab91edf9c917e586b851abcca902f5a55db8a713fb2de6fbd63fe0269eea667af0211bf01d3c04c30e193de96d7d77b32c294c5dfd376ca86dff651881a0de8f32aed2ad486c855ab727d0e185b83a798c18bfd558e11e83c7c9f6b07383d5e7d2594d5751a9b75df9eff4590203010001

        Miner_Solution:
        ['K

        Block_Hash:
        0000ba4b0886818b397598597ae549d707d47f1459138c2fcb900fdf2950568e

        New Block:

        Previous_Block_Hash:
        0000ba4b0886818b397598597ae549d707d47f1459138c2fcb900fdf2950568e

        To:
        30819f300d06092a864886f70d010101050003818d00308189028181009 0b0d59dcddfd2d4e2b879251a4bc24eb5542e8258ecb310c79c009e752429732f5f3e76d72f8f2f4d122fa1b090109637b9f99ea71fe105f376ebf0eba0f2ed66dd12919103c54e62bc3677b3c148c9670f20fe26d40c04535db0c19568e062d81d7f9e37c7653c8e75a10d5f93003dcbb0cbade46c23d77f7a861cf3e0a51ebb0203010001

        From:
        30819f300d06092a864886f70d010101050003818d00308189028181009 0f1de4c291970420e8728c0e23a9737c20d2b68ab91edf9c917e586b851abcca902f5a55db8a713fb2de6fbd63fe0269eea667af0211
```

```
    bf01d3c04c30e193de96d7d77b32c294c5dfd376ca86dff651881a0de8f32aed2ad486c855ab727d0e185b8
    3a798c18bfd558e11e83c7c9f6b07383d5e7d2594d5751a9b75df9eff4590203010001
138.
139.        Amount:
140.        3000
141.
142.        Hash:
143.        2d0cce04e54d257a157f16b0d42c77e1076ebd114c49479b1163e24ae54ea5e4
144.
145.        Signature:
146.        [B@5171d6fa
147.
148.        Inputs:
149.        974dadd9665237a50a386904e38804e79842d8969aac563516edeb618bcad8ba 10
150.        de5f04609e97985626e9b8ab294272cd276dcdf83031b4b7b9eb2754042ac007 91010.0
151.
152.        Outputs:
153.        2d0cce04e54d257a157f16b0d42c77e1076ebd114c49479b1163e24ae54ea5e4 88020.0
154.        2d0cce04e54d257a157f16b0d42c77e1076ebd114c49479b1163e24ae54ea5e4 3000.0
155.
156.        Miner_Reward_Public_Key:
157.        30819f300d06092a864886f70d010101050003818d0030818902818100 9bda7757642474dc726b91
    4db3cb098f30e36fb9dcd383437b1d776b871bb741d5fca7f762b9d7715ba91e3553302a83821c5302e783f
    c3bbf092cbc65195426c6004dfc472395bb376a6500bf609d7ca7256dcbd6704991407034b010f27192c3f6
    10e924c461316fed6f1c8669520df00f6dd86cda28970be3bf330d416ed70203010001
158.
159.        Miner_Solution:
160.        v h
161.
162.        Block_Hash:
163.        00000bf09e61f44449f91be613a0db6c87a5db1f24ed127a4a8a54ac901c808c
```

Appendix B – GenerateKeys.java

```
1.  /**************************************************************************
2.    * Name: Dominic Whyte
3.    *
4.    * Code modified from:
5.    * https://javadigest.wordpress.com/2012/08/26/rsa-encryption-example/
6.    * http://stackoverflow.com/questions/1709441/generate-rsa-key-pair-and-encode-
    private-as-string
7.    *
8.    * Description: Creates a private and public key with the file names specified
9.    * by the first two command line arguments. Also prints out the public key in
10.   * hexadecimal
11.   *
12.   *
13.   *************************************************************************/
14.
15. import java.io.File;
16. import java.io.FileInputStream;
17. import java.io.FileOutputStream;
18. import java.io.ObjectInputStream;
19. import java.io.ObjectOutputStream;
20. import java.security.KeyPair;
21. import java.security.KeyPairGenerator;
22. import java.security.PrivateKey;
23. import java.security.PublicKey;
```

```java
import javax.crypto.Cipher;

/**
 * @author JavaDigest
 *
 */
public class GenerateKeys {

  /**
   * String to hold name of the encryption algorithm.
   */
  public static final String ALGORITHM = "RSA";

  /**
   * String to hold the name of the private key file.
   */
  public static String PRIVATE_KEY_FILE;

  /**
   * String to hold name of the public key file.
   */
  public static String PUBLIC_KEY_FILE;

  /**
   * Generate key which contains a pair of private and public key using 1024
   * bytes. Store the set of keys in Prvate.key and Public.key files.
   *
   * @throws NoSuchAlgorithmException
   * @throws IOException
   * @throws FileNotFoundException
   */
  public static void generateKey() {
    try {
      final KeyPairGenerator keyGen = KeyPairGenerator.getInstance(ALGORITHM);
      keyGen.initialize(1024);
      final KeyPair key = keyGen.generateKeyPair();

      File privateKeyFile = new File(PRIVATE_KEY_FILE);
      File publicKeyFile = new File(PUBLIC_KEY_FILE);

      // Create files to store public and private key
      if (privateKeyFile.getParentFile() != null) {
        privateKeyFile.getParentFile().mkdirs();
      }
      privateKeyFile.createNewFile();

      if (publicKeyFile.getParentFile() != null) {
        publicKeyFile.getParentFile().mkdirs();
      }
      publicKeyFile.createNewFile();

      // Saving the Public key in a file
      ObjectOutputStream publicKeyOS = new ObjectOutputStream(
          new FileOutputStream(publicKeyFile));
      publicKeyOS.writeObject(key.getPublic());
      publicKeyOS.close();

      // Saving the Private key in a file
      ObjectOutputStream privateKeyOS = new ObjectOutputStream(
          new FileOutputStream(privateKeyFile));
      privateKeyOS.writeObject(key.getPrivate());
```

```
85.       privateKeyOS.close();
86.     } catch (Exception e) {
87.       e.printStackTrace();
88.     }
89.
90.   }
91.
92.   /**
93.    * The method checks if the pair of public and private key has been generated.
94.    *
95.    * @return flag indicating if the pair of keys were generated.
96.    */
97.   public static boolean areKeysPresent() {
98.
99.     File privateKey = new File(PRIVATE_KEY_FILE);
100.          File publicKey = new File(PUBLIC_KEY_FILE);
101.
102.          if (privateKey.exists() && publicKey.exists()) {
103.            return true;
104.          }
105.          return false;
106.        }
107.
108.        /**
109.         * Encrypt the plain text using public key.
110.         *
111.         * @param text
112.         *          : original plain text
113.         * @param key
114.         *          :The public key
115.         * @return Encrypted text
116.         * @throws java.lang.Exception
117.         */
118.        public static byte[] encrypt(String text, PublicKey key) {
119.          byte[] cipherText = null;
120.          try {
121.            // get an RSA cipher object and print the provider
122.            final Cipher cipher = Cipher.getInstance(ALGORITHM);
123.            // encrypt the plain text using the public key
124.            cipher.init(Cipher.ENCRYPT_MODE, key);
125.            cipherText = cipher.doFinal(text.getBytes());
126.          } catch (Exception e) {
127.            e.printStackTrace();
128.          }
129.          return cipherText;
130.        }
131.
132.        /**
133.         * Decrypt text using private key.
134.         *
135.         * @param text
136.         *          :encrypted text
137.         * @param key
138.         *          :The private key
139.         * @return plain text
140.         * @throws java.lang.Exception
141.         */
142.        public static String decrypt(byte[] text, PrivateKey key) {
143.          byte[] dectyptedText = null;
144.          try {
145.            // get an RSA cipher object and print the provider
```

```
146.              final Cipher cipher = Cipher.getInstance(ALGORITHM);
147.
148.              // decrypt the text using the private key
149.              cipher.init(Cipher.DECRYPT_MODE, key);
150.              dectyptedText = cipher.doFinal(text);
151.
152.            } catch (Exception ex) {
153.              ex.printStackTrace();
154.            }
155.
156.            return new String(dectyptedText);
157.          }
158.
159.
160.          /**
161.           * Test the EncryptionUtil
162.           */
163.          public static void main(String[] args) {
164.
165.            try {
166.                //Take first two arguments to be part of key name
167.                 PRIVATE_KEY_FILE = "C:/keys/private_" + args[0] + ".key";
168.                 PUBLIC_KEY_FILE = "C:/keys/public_" + args[1] + ".key";
169.
170.                // Method generates a pair of keys using the RSA algorithm and stores it
                    // in their respective files
171.                generateKey();
172.
173.
174.              final String originalText = "Text to be encrypted ";
175.              ObjectInputStream inputStream = null;
176.
177.              // Encrypt the string using the public key
178.              inputStream = new ObjectInputStream(new FileInputStream(PUBLIC_KEY_FILE));
179.
180.              final PublicKey publicKey = (PublicKey) inputStream.readObject();
181.              final byte[] cipherText = encrypt(originalText, publicKey);
182.
183.              //Print out the user's public key
184.              byte[] publicKeyBytes = publicKey.getEncoded();
185.                StringBuffer retString = new StringBuffer();
186.                for (int i = 0; i < publicKeyBytes.length; ++i) {
187.                    retString.append(Integer.toHexString(0x0100 + (publicKeyBytes[i] & 0
    x00FF)).substring(1));
188.                }
189.                System.out.println("Your public key is: ");
190.                System.out.println(retString);
191.
192.
193.              // Decrypt the cipher text using the private key.
194.              inputStream = new ObjectInputStream(new FileInputStream(PRIVATE_KEY_FILE))
    ;
195.              final PrivateKey privateKey = (PrivateKey) inputStream.readObject();
196.              final String plainText = decrypt(cipherText, privateKey);
197.
198.              // Printing the Original, Encrypted and Decrypted Text
199.              System.out.println("Original: " + originalText);
200.              System.out.println("Encrypted: " +cipherText.toString());
201.              System.out.println("Decrypted: " + plainText);
202.
```

```
            } catch (Exception e) {
              e.printStackTrace();
            }
          }
        }
```

Appendix C – PrepareTransaction.java

```
/*****************************************************************************
  * Name: Dominic Whyte
  *
  * Code modified from:
  * https://javadigest.wordpress.com/2012/08/26/rsa-encryption-example/
  * http://stackoverflow.com/questions/1709441/generate-rsa-key-pair-and-encode-
private-as-string
  *
  * Description: Takes the following command line arguments (in this order):
  * receiving public key in hex, name of file containing private key, name of
  * file containing public key, amount of coin to be transferred
  *
  *
  *****************************************************************************/

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import javax.crypto.Cipher;

/**
 * @author JavaDigest
 *
 */
public class PrepareTransaction {

  /**
   * String to hold name of the encryption algorithm.
   */
  public static final String ALGORITHM = "RSA";

  /**
   * String to hold the name of the private key file.
   */
  public static String PRIVATE_KEY_FILE;

  /**
   * String to hold name of the public key file.
   */
  public static String PUBLIC_KEY_FILE;

  /**
   * Generate key which contains a pair of private and public key using 1024
   * bytes. Store the set of keys in Prvate.key and Public.key files.
```

```
50.    *
51.    * @throws NoSuchAlgorithmException
52.    * @throws IOException
53.    * @throws FileNotFoundException
54.    */
55.   public static void generateKey() {
56.     try {
57.       final KeyPairGenerator keyGen = KeyPairGenerator.getInstance(ALGORITHM);
58.       keyGen.initialize(1024);
59.       final KeyPair key = keyGen.generateKeyPair();
60.
61.       File privateKeyFile = new File(PRIVATE_KEY_FILE);
62.       File publicKeyFile = new File(PUBLIC_KEY_FILE);
63.
64.       // Create files to store public and private key
65.       if (privateKeyFile.getParentFile() != null) {
66.         privateKeyFile.getParentFile().mkdirs();
67.       }
68.       privateKeyFile.createNewFile();
69.
70.       if (publicKeyFile.getParentFile() != null) {
71.         publicKeyFile.getParentFile().mkdirs();
72.       }
73.       publicKeyFile.createNewFile();
74.
75.       // Saving the Public key in a file
76.       ObjectOutputStream publicKeyOS = new ObjectOutputStream(
77.           new FileOutputStream(publicKeyFile));
78.       publicKeyOS.writeObject(key.getPublic());
79.       publicKeyOS.close();
80.
81.       // Saving the Private key in a file
82.       ObjectOutputStream privateKeyOS = new ObjectOutputStream(
83.           new FileOutputStream(privateKeyFile));
84.       privateKeyOS.writeObject(key.getPrivate());
85.       privateKeyOS.close();
86.     } catch (Exception e) {
87.       e.printStackTrace();
88.     }
89.
90.   }
91.
92.   /**
93.    * The method checks if the pair of public and private key has been generated.
94.    *
95.    * @return flag indicating if the pair of keys were generated.
96.    */
97.   public static boolean areKeysPresent() {
98.
99.     File privateKey = new File(PRIVATE_KEY_FILE);
100.          File publicKey = new File(PUBLIC_KEY_FILE);
101.
102.          if (privateKey.exists() && publicKey.exists()) {
103.            return true;
104.          }
105.          return false;
106.        }
107.
108.        /**
109.         * Encrypt the plain text using public key.
110.         *
```

```
111.          * @param text
112.          *          : original plain text
113.          * @param key
114.          *          :The public key
115.          * @return Encrypted text
116.          * @throws java.lang.Exception
117.          */
118.         public static byte[] encrypt(String text, PrivateKey key) {
119.           byte[] cipherText = null;
120.           try {
121.             // get an RSA cipher object and print the provider
122.             final Cipher cipher = Cipher.getInstance(ALGORITHM);
123.             // encrypt the plain text using the public key
124.             cipher.init(Cipher.ENCRYPT_MODE, key);
125.             cipherText = cipher.doFinal(text.getBytes());
126.           } catch (Exception e) {
127.             e.printStackTrace();
128.           }
129.           return cipherText;
130.         }
131.
132.         /**
133.          * Decrypt text using private key.
134.          *
135.          * @param text
136.          *          :encrypted text
137.          * @param key
138.          *          :The private key
139.          * @return plain text
140.          * @throws java.lang.Exception
141.          */
142.         public static String decrypt(byte[] text, PublicKey key) {
143.           byte[] dectyptedText = null;
144.           try {
145.             // get an RSA cipher object and print the provider
146.             final Cipher cipher = Cipher.getInstance(ALGORITHM);
147.
148.             // decrypt the text using the private key
149.             cipher.init(Cipher.DECRYPT_MODE, key);
150.             dectyptedText = cipher.doFinal(text);
151.
152.           } catch (Exception ex) {
153.             ex.printStackTrace();
154.           }
155.
156.           return new String(dectyptedText);
157.         }
158.
159.
160.         /**
161.          * Test the EncryptionUtil
162.          */
163.         public static void main(String[] args) {
164.
165.           try {
166.               //receiving public key in hex
167.               String receiver = args[0];
168.               StdOut.print("To: ");
169.
170.               StdOut.print(receiver);
171.               //Take in command line arguments denoting file locations of private
```

```
172.                //and public key of coin sender
173.                 PRIVATE_KEY_FILE = "C:/keys/private_" + args[1] + ".key";
174.                 PUBLIC_KEY_FILE = "C:/keys/public_" + args[2] + ".key";
175.
176.             ObjectInputStream inputStream = null;
177.
178.             // Encrypt the string using the public key
179.             inputStream = new ObjectInputStream(new FileInputStream(PUBLIC_KEY_FILE));

180.             final PublicKey publicKey = (PublicKey) inputStream.readObject();
181.             //final byte[] cipherText = encrypt(originalText, publicKey);
182.
183.             //Print out the user's public key
184.             byte[] publicKeyBytes = publicKey.getEncoded();
185.               StringBuffer retString = new StringBuffer();
186.               for (int i = 0; i < publicKeyBytes.length; ++i) {
187.                   retString.append(Integer.toHexString(0x0100 + (publicKeyBytes[i] & 0
    x00FF)).substring(1));
188.               }
189.               StdOut.print(" From: ");
190.
191.               StdOut.print(retString);
192.
193.               //Print out amount of coin to be transmitted
194.               StdOut.print(" Amount: ");
195.
196.               StdOut.print(args[3]);
197.               //Print unencrypted hash
198.               StdOut.print(" Hash: ");
199.
200.
201.
202.               //string with all text from transaction to be hashed
203.               String transactiontext = ("To:" + receiver + "From:" + retString +
204.                                         "Amount:" + args[3]);
205.               //hash the transactiontext with Sha256
206.               String hashedtransactiontext = Sha256.hash(transactiontext);
207.               StdOut.print(hashedtransactiontext);
208.                               //Print hash encrypted with private key of the user
209.               StdOut.print(" Signature: ");
210.
211.             // Decrypt the cipher text using the private key.
212.             inputStream = new ObjectInputStream(new FileInputStream(PRIVATE_KEY_FILE))
    ;
213.             final PrivateKey privateKey = (PrivateKey) inputStream.readObject();
214.
215.             final byte[] cipherText = encrypt(hashedtransactiontext, privateKey);
216.             //final String plainText = decrypt(cipherText, publicKey);
217.             // Printing the Original, Encrypted and Decrypted Text
218.             System.out.print(cipherText.toString());
219.             //System.out.println("Decrypted: " + plainText);
220.
221.           } catch (Exception e) {
222.             e.printStackTrace();
223.           }
224.         }
225.       }
```

Appendix D – Sha256.java

```
/*****************************************************************************
  * Name: Dominic Whyte
  *
  * Code modified from:
  * http://www.mkyong.com/java/java-sha-hashing-example/
  * Help received from my Computer Science preceptor Dan Leyzberg with the “catching”
  * part of the program (this had not yet been taught in COS126)
  *
  * Description: Outputs the SHA-256 hash of a given String
  *
  *
  *****************************************************************************/
import java.security.MessageDigest;

public class Sha256
{
    public static String hash(String text)
    {
        MessageDigest md;

        // get SHA-256 from MessageDigest
        try {
            md = MessageDigest.getInstance("SHA-256");
        } catch (java.security.NoSuchAlgorithmException e) {
            System.err.println("No such algorithm SHA-256!" + e.getMessage());
            return null;
        }

        String password = text;
        md.update(password.getBytes());
        byte byteData[] = md.digest();

        //convert the byte to hex format method 1
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < byteData.length; i++) {
            sb.append(Integer.toString((byteData[i] & 0xff) + 0x100, 16).substring(1));

        }
        return sb.toString();

    }

    //main for testing
    public static void main(String[] args) {
        System.out.println(hash("testhash"));
    }
}
```

Appendix E – Block.java

```
/*****************************************************************************
  * Name: Dominic Whyte
  *
  * Description: Block object with necessary specifications. Main() takes the
  * hash of the previous block and a prepared transaction from PrepareTransaction
```

```
   * followed by the public key of the miner and creates a Block with these
   * arguments. Also takes Standard Input from ledger.txt file. Make sure to
   * update your ledger and new Blockchain accordingly! Note: if you are sending
   * money to a new account, make sure you first create the account in the ledger
   * by putting their public key under "Public_Key:" etc.
   *
   *************************************************************************/

public class Block {
    //Symbol table with public key as key and String array storing details of
    //unused transactions as the value
    private ST<String, Queue<String[]>> ledger;
    private String previousblock; //the hash of the previous block in blockchain
    private String to;
    private String from;
    private String amount;
    private String hash;
    private String signature;
    private String minerrewardkey; //public key of the miner to receive reward
    private static final int MINING_DIFFICULTY = 4;
    private static final Integer MINING_REWARD = 10;

    //constructor to create Block object
    public Block(String previousblock, String to, String from, String amount,
                 String hash, String signature, String minerrewardkey) {
        this.previousblock = previousblock; //store as instance variable
        //store instance variables
        this.to = to;
        this.from = from;
        this.amount = amount;
        this.hash = hash;
        this.signature = signature;
        this.minerrewardkey = minerrewardkey;

        //Read from StdIn file Ledger.txt the latest accurate ledger. From this
        //ledger, initialize and fill the ledger symbol table
        ledger = new ST<String, Queue<String[]>>(); //intialize ST
        StdIn.readString(); //read "Public Key:"
        while(!StdIn.isEmpty()) {
            String key = StdIn.readString(); //read the public key String
            StdIn.readString(); //read "Unused Transactions:"
            //make a new queue of String[] for this key
            Queue<String[]> transactions = new Queue<String[]>();
            Boolean moveon = false; //true when it is time to move to next key
            while(!moveon) {
                if(StdIn.isEmpty()) {
                    moveon = true;
                }
                else {
                    String next = StdIn.readString();
                    //end loop if all transactions for this key have been added
                    if(next.matches("Public_Key:")) {
                        moveon = true;
                    }
                    else {
                        //make a new String[] with transaction ID as first item
                        //and transaction coin value as second item
                        String[] unusedtrans = new String[2];
                        unusedtrans[0] = next; //the transaction ID
                        unusedtrans[1] = StdIn.readString(); //coin value
                        //enqueue the newest transaction array
```

```
                        transactions.enqueue(unusedtrans);
                    }
                }
            }
            //put into the ledger the public key and all its unused
            //transactions
            ledger.put(key, transactions);
        }
    }
    //method to print out the ledger
    public void printLedger() {
        for (String key : ledger) {
            System.out.println("Key is: " + key);
            for(String[] transactions : ledger.get(key)) {
                System.out.println("Hash is: " + transactions[0] + " with value of " +
transactions[1]);
            }
        }


    }
    //method to print out the ledger
    public void updateLedger() {
        for (String key : ledger) {
            StdOut.println("Public_Key:");
            StdOut.println(key);
            StdOut.println("Unused_Transactions:");
            for(String[] transactions : ledger.get(key)) {
                StdOut.println(transactions[0] + " " + transactions[1]);
            }
        }


    }
          //verify the signature and verify the hash
          public Boolean authenticate() {
              //applying 'from' on the signature should yield the hash

              //also verify the hash for the transaction
              //String with all text from transaction to be hashed
              String transactiontext = ("To:" + to + "From:" + from +
                                        "Amount:" + amount);
              //hash the transactiontext with Sha256
              String hashedtransactiontext = Sha256.hash(transactiontext);
              if (!hashedtransactiontext.matches(hash)) {
                  return false;
              }
              return true;


          }

          //method to print out the new block
          public void printBlock() {
              //Print out instance variables
              System.out.println("Previous_Block_Hash:");
              System.out.println(this.previousblock);
              System.out.println();
              System.out.println("To:");
              System.out.println(this.to);
              System.out.println();
```

```
                System.out.println("From:");
                System.out.println(this.from);
                System.out.println();
                System.out.println("Amount:");
                System.out.println(this.amount);
                System.out.println();
                System.out.println("Hash:");
                System.out.println(this.hash);
                System.out.println();
                System.out.println("Signature:");
                System.out.println(this.signature);
                System.out.println();

                //figure out which transactions will be used as inputs and what the
                //outputs will be (ie. how much is returned to the user)
                Double cost = Double.parseDouble(amount); //amount transaction is for
                //Get the queue from the ledger symbol table with the available funds
                //If 'from' public key has no funds, reject transaction
                if (!ledger.contains(from)) {
                    throw new RuntimeException("Insufficient funds to complete transacti
    on");
                }
                Queue<String[]> availablefunds = ledger.get(from);
                //Make a Queue with transactions used
                Queue<String[]> usedfunds = new Queue<String[]>();
                Boolean paidfor = false; //is the transaction paid for
                double funds = 0.0; //funds taken out of queue
                //repeat until cost has been covered
                while(!paidfor) {
                    //throw an error if there are no more funds
                    if (availablefunds.isEmpty()) {
                        throw new RuntimeException("Insufficient funds to complete trans
    action");
                    }
                    //get the String array with the next unused transaction
                    String[] unusedtransaction = availablefunds.dequeue();
                    funds = funds + Double.parseDouble(unusedtransaction[1]);
                    usedfunds.enqueue(unusedtransaction); //mark as used
                    //if the funds are enough to cover the cost, end loop
                    if (funds >= cost) {
                        paidfor = true;
                    }
                    //else continue getting new transactions to pay cost
                }
                //update ledger
                ledger.put(from, availablefunds);

                //print out which inputs were used
                System.out.println("Inputs:");
                while(!usedfunds.isEmpty()) {
                    String[] transactiontobeused = usedfunds.dequeue();
                    System.out.println(transactiontobeused[0] + " " + transactiontobeuse
    d[1]);
                }
                System.out.println();
                //If the ledger does not contain an entry for the receiver, add it
                if (!ledger.contains(from)) {
                    Queue<String[]> newentry = new Queue<String[]>();
                    ledger.put(from, newentry);
                }
                //Update the ledger with the new transactions and print outputs
```

```
                Double change = funds - cost; //amount to be returned to 'from'
                String[] newtransactionfrom = {hash, change.toString()};
                String[] newtransactionto = {hash, cost.toString()};
                if (change != 0.0) {
                ledger.get(from).enqueue(newtransactionfrom); //add new transaction
                }
                ledger.get(to).enqueue(newtransactionto); //add new transaction
                //print outputs
                System.out.println("Outputs:");
                //By convention, print out the transaction to the 'from' sender first
                System.out.println(newtransactionfrom[0] + " " + newtransactionfrom[1]);

                System.out.println(newtransactionto[0] + " " + newtransactionto[1]);
                System.out.println();
                //Print out public key of the miner (for compensation)
                System.out.println("Miner_Reward_Public_Key:");
                System.out.println(minerrewardkey);
                System.out.println();
                //Give the Miner 10 coins
                //If the Miner does not have any coins, make him an account on the ledge
    r
                String[] reward = {hash, MINING_REWARD.toString()};
                if (!ledger.contains(minerrewardkey)) {
                    Queue<String[]> newfunds = new Queue<String[]>();
                    newfunds.enqueue(reward);
                    ledger.put(minerrewardkey, newfunds);
                }
                else {ledger.get(minerrewardkey).enqueue(reward);}
                //Mining problem solution
                //inputs ommitted in blocktext for simplicity
                String blocktext = ("Previous_Block_Hash:" + this.previousblock + "To:"
    +
                                    to + "From:" + from + "Amount:" + amount + "Hash:" +

                                    hash + "Signature:" + signature + "Outputs:" +
                                    newtransactionfrom[0] + " " + newtransactionfrom[1]
    +
                                    newtransactionto[0] + " " + newtransactionto[1] +
                                    "Miner_Reward_Public_Key:" + minerrewardkey);
                System.out.println("Miner_Solution:"); //solution to mining problem
                String solution = Miner.findkey(MINING_DIFFICULTY, blocktext);
                System.out.println(solution);
                System.out.println();
                System.out.println("Block_Hash:");
                System.out.println(Sha256.hash(blocktext + solution));
                System.out.println();

            }
            //main method for testing
            public static void main(String[] args) {
                String previous = args[0]; //previous block hash
                String to = args[2]; //"to" public key
                String from = args[4]; //"from" public key
                String amount = args[6]; //amount transaction is for
                String hash = args[8]; //hash
                String signature = args[10]; //signature
                String minerrewardkey = args[11]; //public key of miner for compensation

                Block block = new Block(previous, to, from, amount, hash, signature, min
    errewardkey);
```

```
                //check signature and hash
                if (block.authenticate())
                    System.out.println("Authentication Success");
                else
                    System.out.println("Authentication Failure");
                if (block.authenticate()) {
                    System.out.println("New Block:");
                    System.out.println();
                    block.printBlock();
                    //update ledger only if transaction was verified

                    System.out.println("Updated Ledger:");
                    System.out.println();
                    block.updateLedger();
                }
            }
        }
```

Appendix F – Miner.java

```
/**********************************************************************
 * Name: Dominic Whyte
 *
 * Description: Mines for a key which, when concatenated to the end of a given
 * String, can be hashed using SHA-256 to yield a String with N leading zeros
 *
 *
 **********************************************************************/

public class Miner {
    //checks if a given String has at least N leading zeros
    public static Boolean checksuccess(int N, String key) {
        //check the first N digits and if any of them are not zero (char 48),
        //return false. Else return true
        for(int i = 0; i < N; i++){
            if (((int) key.charAt(i)) != 48) {
                return false;
            }
        }
        return true;

    }
    //takes number of required leading zeros N and String code to which key will
    //be concatenated with and returns the key which yields a code + key
    //concatenation with at least N leading zeros
    //Warning: this will take a considerable amount of time depending on how
    //large N is (as it is supposed to: this is a "proof of work")
    public static String findkey(int N, String code) {
        Boolean success = false; //Has the right hash been found?
        String key = ""; //the current key being tested for success
        int i = 0; //the trial number
        //run until success has been achieved
        //a list of ascii characters
        String ascii = " !#$%&\'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~";
        while(!success) {
            i++; //increment which trial we are on
            int digits = 1; //how long the test key is
```

```
            //digits should be 1 for ascii.length() times, 2 for (ascii.length())
            //squared times, etc. (so you try more random keys with more random
            //digits)
            boolean bool = true;
            while(bool) {
                //if ascii.length()^digits < i, then increment digits
                //Use change of log base rule for this
                if (((double) Math.log(i)/Math.log((double)ascii.length())) > digits) {

                    digits++;
                }
                else
                    bool = false;
            }
            //set the key to be a random String from ascii alphabet with length
            //digits
            key = RandomString.randomString(digits);
            //If the concatenation of the code and the possible key yields a
            //hash with at least N leading zeros, deem the key a success
            if(checksuccess(N, Sha256.hash(code + key))) {
                success = true;
            }
        }
        //StdOut.println("Trials: " + i);
        return key;

    }

    public static void main(String[] args) {
        //start timer
        Stopwatch timer = new Stopwatch();
        //number of leading zeros required
        int N = Integer.parseInt(args[0]);
        //code to which key will be concatenated with
        String code = args[1];
        //print out the key found by the method findkey
        System.out.println(findkey(N, code));
        //print time taken
        System.out.println("Time elapsed: " + timer.elapsedTime());
    }
}
```

Appendix G – RandomString.java

```
/*************************************************************************
  * Name: Dominic Whyte
  *
  * Code modified from:
  * http://stackoverflow.com/questions/41107/how-to-generate-a-random-alpha-numeric-
string
  *
  * Description: Outputs the SHA-256 hash of a given String
  *
  *
  *************************************************************************/

import java.util.Random;
public class RandomString {
```

```
14.     //ascii alphabet
15.     static final String AB = " !#$%&\'()*+,-
    ./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~";
16.     static Random rnd = new Random();
17.     public static String randomString(int len){
18.         StringBuilder sb = new StringBuilder( len );
19.         for( int i = 0; i < len; i++ )
20.             sb.append( AB.charAt( rnd.nextInt(AB.length()) ) );
21.         return sb.toString();
22.     }
23.     //tester method
24.     public static void main(String[] args) {
25.
26.         StdOut.println(randomString(3));
27.
28.     }
29. }
```