



# *Advanced sockets programming*

- 5.1 Object serialization
- 5.2 Objects over Sockets

- 5.3 Digitally signed messages
- 5.4 Summary

The distributed applications we have developed so far sent only data streams over sockets. What if you want to transfer full-blown objects over the wire? Object serialization is a technique by which a program can save the state of objects to a file and later on read them back into memory or send an object over the network.

In this chapter, you'll learn how object serialization can be used to send object over sockets and implement persistency for Java objects. Then we'll discuss authenticity—methods, such as digital signatures, for determining whether a message really originated from the client that the message claims it came from.

## **5.1 OBJECT SERIALIZATION**

Object serialization is a mechanism used extensively in RMI, as you will see in part II of this book. However, it is also useful in any program that wants to save the state of objects to a file and read those objects later to reconstruct the state of the program, or to send an object over the network using low-level sockets.

In object serialization, the object to be worked with is a Java object that must implement the `java.io.Serializable` interface—if it doesn't, it can't be used with object serializa-

tion. Note that the `Serializable` interface does not have any methods. It is merely used to inform the JVM that you want the object to be serialized.

Besides an object, you also need an I/O stream. To use object serialization for saving objects, you must create an instance of an `ObjectOutputStream`, which is a subclass of `FilterOutputStream`. As an example, the following code segment shows how to save a serialized string to a file:

```
FileOutputStream fos = new FileOutputStream("str.out");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject("this string is being saved");
```

The `writeObject` method is used to save an object to the output stream. This method can be called any number of times to save any number of objects. However, the object passed to `writeObject` must implement the `Serializable` interface.

Reading the objects you have saved is trivial. The following example shows how to read the serialized string object you saved above.

```
FileInputStream fis = new FileInputStream("str.out");
ObjectInputStream ois = new ObjectInputStream(fis);
Object o = ois.readObject();
```

As with `writeObject`, `readObject` can be called any number of times to read any number of objects from the input stream. When you are reading objects from a stream, you must know what type of objects are expected in the stream. This means that programs that serialize objects should be kept in sync with programs that deserialize objects.

### 5.1.1 Object serialization and persistence

In this section we will revisit the problem of object persistence that we discussed in chapter 4 in regards to keeping track of employees' data. We will see how object serialization can provide us with a persistent storage.

First, let's modify our `Employee` class (developed in example 4.1) to include object serialization. We first need to have our class implement the `Serializable` interface to inform the JVM of our intention of serializing objects. Notice that I have modified the `print()` method a bit. The new `Employee` class is shown in example 5.1.

#### Example 5.1: `Employee.java`

```
import java.io.*;
/**
 * @(#)Employee.java
 * A serialized class to keep track of employees
 */
public class Employee implements Serializable {
    String name;
    int age;
    int salary;

    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
    }
}
```

```

        this.salary = salary;
    }

    public void print() {
        System.out.println("Record for: "+name);
        System.out.println("Name: "+name);
        System.out.println("Age: "+age);
        System.out.println("Salary: "+salary);
    }
}

```

The class in example 5.1 does some initialization through a constructor and provides a method for printing an employee's records.

Now we can develop a new class that creates instances, or objects, of `Employee` and serializes them by saving their states in a file we'll call `db`. Example 5.2 shows this new class.

### Example 5.2: SaveEmp.java

```

import java.io.*;
/**
 * @(#)SaveEmp.java
 * This class creates some instances of the Employee
 * class and serializes them by saving their states to
 * a file.
 */
public class SaveEmp {
    public static void main(String argv[]) throws Exception {
        // Create some objects.
        Employee emily = new Employee("E. Jordan", 27, 35000);
        Employee john = new Employee("J. McDonald", 290, 39000);
        // Serialize the objects emily and john.
        FileOutputStream fos = new FileOutputStream("db");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(emily);
        oos.writeObject(john);
        oos.flush();
    }
}

```

Running example 5.2 creates two objects, `emily` and `john`, and serializes them by saving their states to a file called `db`.

Now we can write a new class that deserializes the objects `emily` and `john` and prints their records. The new class is shown in example 5.3.

### Example 5.3: ReadEmp.java

```

import java.io.*;
/**
 * @(#)ReadEmp.java
 * This class deserializes the objects emily and john by reconstructing
 * their states, which are saved in a file.
 */

```

```

public class ReadEmp {
    public static void main(String argv[]) throws Exception{
        // Deserialize the objects emily and john.
        FileInputStream fis = new FileInputStream("db");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Employee emily = (Employee) ois.readObject();
        Employee john = (Employee) ois.readObject();
        // Print the records after reconstructing the states.
        emily.print();
        john.print();
    }
}

```

Compile and run the code in example 5.4 to see how the program will reconstruct the state of the objects `emily` and `john`, and how it prints their data.

### 5.1.2 Security in object serialization

To see the important benefit of the `Serializable` interface, let us consider the following code:

```

public class PasswordFile implements Serializable {
    private String passwd;
    . . . . .
}

```

If we serialize this object, we will end up writing the password to the file, among other data, and anyone will be able to read the password. This happens because object serialization has access to all instance variables, including `private`, within a serializable class.

There are two ways to serialize an object without exposing any sensitive data to the world. The first is to mark any sensitive data fields as `transient`. The `passwd` above can now be written as:

```
private transient String passwd;
```

At the time serialization happens, the JVM will skip over any fields that are declared `transient`. However, if you want to serialize the transient data with the rest of the object, you must override the `writeObject` and `readObject` methods, in which you can control what data is sent and how it is stored.

The second way for dealing with sensitive data is to implement the `Externalizable` interface, which is a subclass of the `Serializable` interface. This interface has two methods that a class must implement: the `writeExternal` and `readExternal` methods.

### 5.1.3 Controlling the serialization

So far, serialization has been performed automatically using `ObjectInputStream`, `ObjectOutputStream`, `writeObject`, and `readObject`. However, there are situations, as you saw earlier with the `PasswordFile`, where you might want to have control over which fields are serialized and which are not. In addition to the `Serializable` interface we have used, the `java.io` package includes another interface: `Externalizable`. This inter-

face is to be used when you want to define an object that has a complete control over serialization (such as the encoding used to send data).

The `Externalizable` interface, which is a subclass of `Serializable`, has two methods that an `Externalizable` object must implement. The two methods are `writeExternal` and `readExternal`, and they have the following signatures:

```
public void writeExternal(ObjectOutput) throws IOException
```

```
public void readExternal(ObjectInput) throws IOException
```

Implementing these methods is not a complicated process. It really depends on how you want to encode the information encapsulated in an object. For example, you may want to use some cryptographic routines to encode the data. When implementing the `writeExternal` method, you must manually write each field of data that you want to save to the `ObjectOutput`. The implementation of `readExternal` is just undoing what `writeExternal` did.

As an example, let me show you how the `PasswordFile` class can be serialized to protect the `passwd` field from being read by the world by implementing the `Externalizable` interface. As I mentioned above, the first step is to implement the `writeExternal` method. Here is a sample implementation:

```
public void writeExternal(ObjectOutput objout) throws IOException {
    // Some other code goes here.

    // Write the field.
    objout.writeObject(passwd);

    // Do other things.
}
```

Now we need to decode the object from the `ObjectInput` in the `readExternal` method, which can be implemented as follows:

```
public void readExternal(ObjectInput objin) throws IOException, ClassNotFoundException {
    // Read the field.
    this.passwd = (String) objin.readObject();
    // Do other things.
}
```

#### 5.1.4 Versioning serialized objects

If you write a useful piece of software, then most likely you will be the one providing maintenance for it. Maintenance may include bug fixes and adding new features. Every time you have a new version of your classes, you will have to think about backward compatibility. As an example, if we have a serialized object and you want to remove an instance variable or add a new one, what would be the effect if you want to reserialize or deserialize?

The object serialization mechanism uses an identifier to keep track of all classes. This identifier is called the `serialVersionUID`, and it is computed from the structure of the class to form a unique 64-bit value identifier. The JDK comes with a program called `serialver` that can be used to find out if a class is serializable and that can get its `serialVersionUID`.

If you invoke the program with `-show`, it puts up a simple user interface as the one shown in figure 5.1.

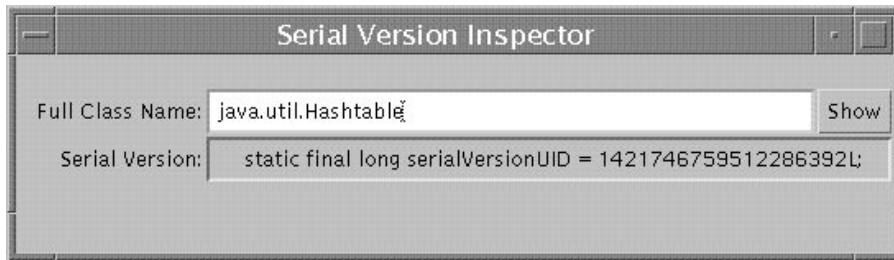


Figure 5.1 The `serialVersionUID` for `java.util.Hashtable`

To find out if a class is serializable and to determine its `serialVersionUID`, enter its full class name in the input text field and click the Show button. If the class is serializable, you will get a `serialVersionUID` value. Figure 5.1 shows the `serialVersionUID` for `java.util.Hashtable`. Once you get this value, you can include it in subsequent versions of the class to indicate that the new version of the class is compatible with the version identified by the `serialVersionUID`. This can be done by declaring the following in your version class:

```
Static final long serialVersionUID = 1421746759512286392L;
```

Note that the value computed by `serialver` is fixed for all compatible classes.

## 5.2 OBJECTS OVER SOCKETS

The classes included in the `java.net` package contain methods for reading/writing (or exchanging) primitive data types between client and server processes. As you saw in chapter 3 when we developed the arithmetic server, we had to write our own methods to read and write arrays of integers to sockets since no methods are provided for such operations. A good question at that time would be whether we can transfer objects over sockets. The answer is definitely yes. With the object serialization API, it is possible to write objects over sockets.

### 5.2.1 Sending objects on the wire

If you are already familiar with distributed object technology, you may ask why we are using sockets for this. Why not use a distributed object system (such as RMI) for this purpose? If you would like to send objects over sockets while avoiding the overhead of using a distributed object system, then using sockets and object serialization is the way to go.

In this section I will show you how to send objects over sockets using the example we developed in chapter 3—the arithmetic server. As a reminder of what that server does, the client sends two arrays of integers to the server, which then adds up the arrays and returns the result back to the client.

*Developing a serializable class* We need a class that would serve as an interface for the client and server. This class will define two methods: one for setting the array and one for displaying the result. Example 5.4 shows the `MathObj` class.

#### Example 5.4: `MathObj.java`

```
import java.io.*;
import java.util.*;
/**
 * @(#)MathObj.java
 */
public class MathObj implements Serializable {
    private int x[] = null;
    public MathObj(){
    }

    public int[] set(int msg[]){
        x = msg;
        return x;
    }

    public int[] out() {
        return x;
    }
}
```

The most important thing to note about the `MathObj` class is that it implements the `Serializable` interface. Again, the `Serializable` interface does not have any methods—it is merely used to let the JVM know that you want to allow the object to be serialized. The methods defined in the `MathObj` class are simple enough. The `set()` method takes an array of integers and returns an array of integers. The `out()` method is used to display the sum of the two arrays.

*Implementing the server* The server is a simple normal server that waits for a connection. When a connection is made, it reads an object from the client. Reading an object is done by invoking the `readObject()` method of the `InputStream`. Notice how the server creates objects of type `MathObj`. Example 5.5 shows the complete source code for the server class.

#### Example 5.5: `ArithServer.java`

```
import java.io.*;
import java.net.*;
import java.util.*;
/**
 * @(#)ArithServer.java
 * This example shows how to use object serialization to send and receive
 * objects over sockets.
 */
public class ArithServer {
    /**
```

```

    * Create the server socket and use its stream to receive serialized
    * objects.
    */
public static void main(String args[]) {
    ServerSocket ser = null;
    Socket soc = null;
    MathObj x = null;
    MathObj y = null;
    int z1[] = new int[5];
    int z2[] = new int[5];
    int result[] = new int[5];
    try {
        ser = new ServerSocket(4343);
        /**
         * This will wait for a connection to be made to this socket.
         */
        soc = ser.accept();
        InputStream o = soc.getInputStream();
        ObjectInput s1 = new ObjectInputStream(o);
        OutputStream o2 = soc.getOutputStream();
        ObjectOutput s2 = new ObjectOutputStream(o2);
        x = (MathObj) s1.readObject();
        y = (MathObj) s1.readObject();
        z1 = x.out();
        z2 = y.out();
        for(int i=0; i<z1.length; i++) {
            System.out.println(z1[i]);
        }
        for(int i=0; i<z2.length; i++) {
            System.out.println(z2[i]);
        }
        for(int p=0; p<z1.length; p++) {
            result[p] = z1[p] + z2[p];
        }
        for(int pl=0; pl<result.length; pl++) {
            System.out.println(result[pl]);
        }
        MathObj myM = new MathObj();
        myM.set(result);
        s2.writeObject(myM);
        s2.flush();
        s1.close();
        s2.close();
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println(e.getMessage());
    }
}
}

```

***Implementing the client*** The client program is, again, a simple client. This client defines two arrays and uses the `MathObj` class's methods to write them to the server. It simply defines

two `MathObj` objects and uses the `set()` method to define the two arrays. It then writes the two instances of `MathObj` to the server as objects using `writeObject`. The client source code is shown in example 5.6.

#### Example 5.6: `ArithClient.java`

```
import java.io.*;
import java.util.*;
import java.net.*;
/**
 * @(#)ArithClient.java
 */
public class ArithClient {
    public static void main(String args[]) {
        int a[] = {4, 4, 4, 4, 4};
        int b[] = {2, 2, 2, 2, 2};

        try {
            // Create a socket.
            Socket soc = new Socket(InetAddress.getLocalHost(), 4343);
            OutputStream o = soc.getOutputStream();
            ObjectOutput s = new ObjectOutputStream(o);
            InputStream in = soc.getInputStream();
            ObjectInput s2 = new ObjectInputStream(in);
            MathObj a1 = new MathObj();
            MathObj a2 = new MathObj();
            MathObj res = null;
            int arr[] = new int[5];

            a1.set(a);
            a2.set(b);
            s.writeObject(a1);
            s.writeObject(a2);
            s.flush();

            res = (MathObj) s2.readObject();
            arr = res.out();
            for(int i=0; i<arr.length; i++) {
                System.out.println(arr[i]);
            }
            s.close();
            s2.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

## 5.3 DIGITALLY SIGNED MESSAGES

The proliferation of world-wide computer networks has led to communication channels protection issues. Communication channels are accessible to eavesdroppers, and the only way to enforce protection of those channels is to apply cryptography. Cryptography is used to protect information to which illegal access is possible.

### 5.3.1 The `java.security` package

The Java security API lets developers incorporate low-level and high-level security functionality into their applications. It provides APIs for digital signatures, message digests, key management, and access control lists. Right now, we are only concerned with digital signatures and how to use them to authenticate messages over sockets.

### 5.3.2 Digital signatures

In real life, it is easy to differentiate between originals and copies. For example, it's easy to tell the difference between a handwritten note and a photocopy of it. In the digital world, however, the task of differentiating between originals and copies is almost impossible because all information is represented as bits. For example, imagine that you, as a client, send a bank transaction to a server, but before your message reaches the server, someone along the wire modifies it so that the reply from the server would go back to him. In the digital world, we need a mechanism by which one party can send a "signed" message to another party so that the receiver can verify the true identity of the sender.

A digital signature establishes sender authenticity. It is analogous to an ordinary written signature in these ways:

- It must be able to identify the author, data, and time of the signature.
- It must be verifiable by third parties to resolve disputes.

For digital signatures to be of practical use, they must have the following properties:

- Easy to produce.
- Easy to recognize and verify.
- Computationally infeasible to forge.

An example of a digital signature is the digital signature algorithm (DSA), which is supported by the security API in JDK 1.1 and above. Digital signatures can be implemented using a simple scheme provided by public-key cryptosystems in which each party has a pair of keys (one is public and the other is private), as opposed to just one private key as in single- or private-key cryptosystems. The interfaces that provide for generating and using digital signatures in JDK1.1 include `KeyGenerator`, `KeyPairGenerator`, and `Signature`, to name a few.

### 5.3.3 Example: signing a file over sockets

The example presented in this section is a client/server application where the client reads a file, generates a pair of keys, signs the contents of the file, and sends an object over the network to a server. The object contains the public key generated by the client, the signature, and the contents of the file. The server receives the object and verifies that the signature is correct.

In this example, three pieces of code need to be developed: a signed object, a client application, and a server application. Let's look at each separately.

*Step 1: Developing a signed object* In order to send objects over sockets, we have to use object serialization. Therefore, we need to serialize our signed object. This can be done easily by having our signed object implement the `Serializable` interface. As mentioned earlier, the `Serializable` interface does not have any methods, so there are no methods to override. Example 5.7 shows the implementation of a `SignedObject`.

#### Example 5.7: SignedObject.java

```
import java.io.*;
import java.security.*;
/**
 * @(#)SignedObject.java
 */
public class SignedObject implements Serializable {
    byte b[];
    byte sig[];
    PublicKey pub;

    // Constructor
    public SignedObject(byte b[], byte sig[], PublicKey pub) {
        this.b = b;
        this.sig = sig;
        this.pub = pub;
    }
}
```

The first thing to note from example 5.7 is the second import statement:

```
import java.security.*;
```

The methods for signing data are contained in the `java.security` package, so we are importing everything from that package. Also, note that the `SignedObject` will carry three things: the contents of a file from the client, a signature, and a public key. `PublicKey` is actually a methodless interface used for type safety and identification for public keys. A public key is really a number associated with an entity (such as an individual or an organization), and everyone who wants to have trusted interactions with that entity should know of it.

*Step 2: Developing a client application* The client in this example has four tasks to perform:

- Read the contents of a file, which are given to it as an argument on the command line.
- Generate a pair of keys (public and private) using the security API.
- Sign the contents of the file.
- Send an object containing the above three items across the network to the server.

These steps can be followed easily from looking at the client's code that is shown in example 5.8.

### Example 5.8: Client.java

```
import java.io.*;
import java.net.*;
import java.security.*;
/**
 * @(#)Client.java
 */
public class Client {
    public static void main(String argv[]) {
        Socket s = null;
        ObjectOutputStream os = null;

        try {
            s = new Socket("purejava", 4000);
            os = new ObjectOutputStream(s.getOutputStream());
            System.out.println("Generating keys...this may take a few minutes");
            // Generate public and private keys.
            KeyPairGenerator kgen = KeyPairGenerator.getInstance("DSA");
            kgen.initialize(256);
            KeyPair kpair = kgen.generateKeyPair();

            // Generate a signature.
            System.out.println("Generating Signature...");
            Signature sig = Signature.getInstance("SHA/DSA");
            PublicKey pub = kpair.getPublic();
            PrivateKey priv = kpair.getPrivate();
            sig.initSign(priv);

            // Read a file and compute a signature.
            FileInputStream fis = new FileInputStream(argv[0]);
            byte arr[] = new byte[fis.available()];
            fis.read(arr);
            sig.update(arr);

            // Send the SignedObject on the wire.
            SignedObject obj = new SignedObject(arr, sig.sign(), pub);
            os.writeObject(obj);

            // Close streams.
            fis.close();
            os.close();
            s.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The two most important things that are accomplished in example 5.8 are generating the public and private keys and signing the data file.

In order to generate a digital signature, we must generate the public and private keys using the `KeyPairGenerator` class. In example 5.8, we are generating a public and private key pair for the algorithm named DSA, which has a length of 256 bits. The next step is to initialize the pair of keys. This is done using the `initialize()` method of the `KeyPairGenerator` class. The argument to the `initialize()` method is the strength of the key in bits. In our example we are generating keys of length 256 bits. Please remember that the longer a key is, the more secure it is. We are using 256 bits in this example so that the computer will generate the keys in a matter of seconds. Longer key lengths require more CPU power to generate the keys. Stronger and more secure keys can be initialized more securely using this line:

```
kgen.initialize(1024, new SecureRandom());
```

In this case, the `initialize()` method takes two arguments: the strength and the source of randomness. The source of randomness must be an instance of the `SecureRandom` class; to keep things simple, we are using an empty constructor of the `SecureRandom` class. In this case, it will automatically generate a “seed” value required for the random number generator.

The next and final step in generating the pair of keys is to use the `KeyPair` class as follows, and as shown in example 5.8:

```
KeyPair pair = kgen.generateKeyPair();
```

At this point we are ready to sign the data:

As shown in example 5.8, a digital signature is created using an instance of the `Signature` class. To sign the data, however, involves four steps:

- 1 *Getting a signature object.* To get a signature object for generating signatures using the DSA algorithm, we use the following fragment of code, as shown in example 5.8:

```
Signature sig = Signature.getInstance("SHA/DSA");
```

- 2 *Initializing the signature object.* Before a signature object can be used for signing, it must be initialized. The initialization is done using the `initSign` method of the `Signature` class. This method takes an instance of `PrivateKey` as an argument. This is done as follows (also shown in example 5.8):

```
PrivateKey priv = kpair.getPrivate();  
sig.initSign(priv)
```

- 3 *Providing the data to be signed to the signature object.* This is done by reading the data (a file in our case) to be signed into an array of bytes and then supplying it to the signature object by calling the `update` method of the `Signature` class as shown in example 5.8.
- 4 *Generating the signature and sending it over the wire.* The last step is to generate the digital signature for the data, and, of course, send it over the wire. In example 5.8, this is shown as follows:

```
SignedObject obj = new SignedObject(arr, sig.sign(), pub);
os.writeObject(obj); // Write the signed object to the output stream.
```

*Step 3: Developing a server application* A server would have to run forever listening for client's requests. In this case, however, the server will also need to read an object from the client and verify that the signature is valid. The server code is shown in example 5.9. The server here does not inherit from the `Thread` class, as discussed in chapter 3; instead, it implements the `Runnable` interface. This is another way of implementing multithreaded servers, as we discussed in chapter 3.

### Example 5.9: Server.java

```
import java.io.*;
import java.net.*;
import java.security.*;
/**
 * @(#)Server.java
 */
public class Server implements Cloneable, Runnable {
    ServerSocket service = null;
    Socket clientSocket = null;
    ObjectInputStream ois = null;
    Thread worker = null;
    KeyPairGenerator kgen;
    KeyPair kpair;

    public static void main(String argv[]) throws IOException {
        Server serv = new Server();
        serv.startServer();
    }

    public synchronized void startServer() throws IOException {
        if (worker == null) {
            service = new ServerSocket(4000);
            worker = new Thread(this);
            worker.start();
        }
    }

    public void run() {
        Socket client = null;
        if (service != null) { // Original or clone?
            while(true) {
                try {
                    client = service.accept();
                    Server newServer = (Server) clone();
                    newServer.service = null;
                    newServer.clientSocket = client;
                    newServer.worker = new Thread(newServer);
                    newServer.worker.start();
                } catch(IOException e) {
```

```

        e.printStackTrace();
    } catch(CloneNotSupportedException e) {
        e.printStackTrace();
    }
}
} else {
    perform(clientSocket);
}
}

private void perform(Socket client) {
    try {
        ois = new ObjectInputStream(clientSocket.getInputStream());
        // Read object from client.
        SignedObject obj = (SignedObject) ois.readObject();
        // Generate object's signature.
        Signature sig = Signature.getInstance("SHA/DSA");
        sig.initVerify(obj.pub);
        sig.update(obj.b);

        // Verify the signature.
        boolean valid = sig.verify(obj.sig);
        if (valid) {
            System.out.println("Signature is valid");
        } else {
            System.out.println("Signature is not valid...spy!");
        }
    } catch(Exception e) {
        e.printStackTrace();
    }

    // Close streams and connection.
    try {
        ois.close();
        clientSocket.close();
    } catch(IOException ex) {
        ex.printStackTrace();
    }
}
}
}

```

Besides listening for connections, the server program in this example will have to receive the signed object and verify if the signature is valid. To verify the authenticity of the signature, we need the data, the signature, and the public key corresponding to the private key used to sign the data.

To start the verification process, just as with signature generation, we need to create an instance of the `Signature` class:

```
Signature sig = Signature.getInstance("SHA/DSA");
```

The next step is to initialize the signature object with the public key:

```
sig.initVerify(obj.pub); // Please refer to example 5.8.
```

Then we'll use the `update()` method to provide the signed data to the signature that we want to verify:

```
sig.update(obj.b);
```

And finally, we can verify the signature and report the result:

```
boolean valid = sig.verify(obj.sig);
if (valid) {
    System.out.println("Signature is valid");
} else {
    System.out.println("Signature is not valid");
}
```

In this client/server example, the flag `valid` should be `true` if we are verifying the signature we generated. If someone was spying over the network and managed to modify the signature, then the flag would be `false`.

## 5.4 SUMMARY

- Object serialization is a mechanism that can be used in any application that needs to save the state of objects to a file and read them back into memory later on, or to send an object over the network using low-level sockets.
- For a Java object to be serialized it must implement the `java.io.Serializable` interface. This interface merely informs the JVM that you want the object to be serialized.
- DSA is a way of generating a digital signature, which makes it possible for a message receiver to verify the claimed identity of the message sender. DSA is supported in JDK1.1 in the `java.security` package.