Almost every application requires some means of keeping data across program runs. Most applications use a file or database for the storage or persistence of data. However, databases are not typically used to store objects, particularly Java objects. On the other hand, flat files alone do not cope well with object structure. What is required is some means to preserve the state of a Java object so that it may be easily stored and subsequently restored to its original state.

Object serialization is a facility that enables objects to be “flattened” out so that they can be stored in a file or sent in a stream across a network. This is accomplished by “writing” the object into an `ObjectOutputStream` instance, which is then used to resurrect the object from the corresponding flattened representation. The serialization classes convert graph (hierarchies) of objects into bytestreams. Serialized objects may be written to a storage device for persistent retention of their state information or shipped across networks for reconstruction on the other side.

The JDK 1.1 (and beyond) provides the Object Serialization mechanism to tackle this once notorious problem of object persistency. Serialization also allows objects to be easily distributed across various Java Virtual Machines (JVMs). As such, we will also discuss Remote Method Invocation (RMI) where a program running on one JVM may invoke methods of objects on another JVM. In this scenario, Java RMI uses the Object Serialization API to pass and return objects during remote method invocation. We will examine RMI and what it brings to Java applications, with an emphasis on understanding the key concepts behind RMI. We will also develop simple applications to illustrate these concepts.

### 16.1 Object Serialization

The design of object serialization allows for most common cases to be handled easily. The following example code in Listing 16-1 shows:
• a Serialize class program that accepts a filename argument, and with methods write() and read() as representative code for serializing operations.

• an ObjectOutputStream being created from an OutputStream instance (in the form of a FileOutputStream object), and writing out via the method writeObject();

• an ObjectInputStream being created from an InputStream instance (in the form of a FileInputStream object), and reading via the method readObject().

```java
import java.util.*;
import java.io.*;

class Serialize {
    String filename;
    public static void main(String[] args) {
        Serialize a = new Serialize(args);
        a.write("This is a Serialization Test");
        System.out.println(a.read());
    }
    public Serialize(String[] args) {
        if (args.length < 1) {
            System.err.println("Usage: Serialize filename");
            System.exit(0);
        } else
            filename = args[0];
    }
    public void write(String str) {
        try {
            FileOutputStream out = new FileOutputStream(filename);
            ObjectOutputStream outobj = new ObjectOutputStream(out);
            outobj.writeObject(str);
            outobj.flush(); outobj.close();
        } catch (Exception e) {
            System.err.println("Failure while writing: "+ e.getMessage());
            e.printStackTrace();
        }
    }
    public String read() {
        try {
            FileInputStream in = new FileInputStream(filename);
            ObjectInputStream inobj = new ObjectInputStream(in);
            String str = (String) inobj.readObject();
            inobj.close();
            return str;
        } catch (Exception e) {
            e.printStackTrace(); return null;
        }
    }
}

Listing 16-1: Serialize.java.