15.5 XML Serialization

In Section 15.4, we demonstrated how to write the individual fields of a record into a file as text, and how to read those fields from a file. Sometimes we want to write an entire object to or read an entire object from a file or—as you'll see in the online REST Web Services chapter—over a network connection. As we mentioned in Chapter 12, XML (eXtensible Markup Language) is a widely used language for describing data. XML is one format commonly used to represent objects. In our online web services chapter, we'll also use another common format called JSON (JavaScript Object Notation) to transmit objects over the Internet. We chose XML in this section rather than JSON, because the APIs for manipulating objects as XML are built into Java SE, whereas the APIs for manipulating objects as JSON are part of Java EE (Enterprise Edition).

In this section, we'll manipulate objects using **JAXB (Java Architecture for XML Binding)**. As you'll see, the JAXB enables you to perform **XML serialization**—which JAXB refers to as **marshaling**. A **serialized object** is represented by XML that includes the object's data. After a serialized object has been written into a file, it can be read from the file and **deserialized**—that is, the XML that represents the object and its data can be used to recreate the object in memory.

15.5.1 Creating a Sequential File Using XML Serialization

The serialization we show in this section is performed with character-based streams, so the result will be a text file that you can view in standard text editors. We begin by creating and writing serialized objects to a file.

Declaring Class `Account`

We begin by defining class `Account` (Fig. 15.9), which encapsulates the client record information used by the serialization examples. All the classes for this example and the one in Section 15.5.2 are located in the `SerializationApps` directory with the chapter's examples. Class `Account` contains `private` instance variables `account`, `firstName`, `lastName` and `balance` (lines 4–7) and *set* and *get* methods for accessing these instance variables. Though the *set* methods do not validate the data in this example, generally they should.

```
1   // Fig. 15.9: Account.java
2   // Account class for storing records as objects.
3   public class Account {
4      private int accountNumber;
5      private String firstName;
6      private String lastName;
7      private double balance;
```

```
                        8
 9      // initializes an Account with default values
 10      public Account() {this(0, "", "", 0.0);}
                        11
 12     // initializes an Account with provided value
 13     public Account(int accountNumber, String firs
   14         String lastName, double balance) {
   15        this.accountNumber = accountNumber;
      16         this.firstName = firstName;
       17         this.lastName = lastName;
        18         this.balance = balance;
                     19      }
                        20
          21      // get account number
 22     public int getAccountNumber() {return account
                        23
          24      // set account number
 25     public void setAccountNumber(int accountNumbe
  26        {this.accountNumber = accountNumber;}
                        27
             28      // get first name
 29     public String getFirstName() {return firstNam
                        30
             31      // set first name
 32     public void setFirstName(String firstName)
     33         {this.firstName = firstName;}
                        34
              35      // get last name
 36     public String getLastName() {return lastName;
                        37
              38      // set last name
 39     public void setLastName(String lastName) {thi
                        40
               41      // get balance
 42     public double getBalance() {return balance;}
                        43
              44      // set balance
 45     public void setBalance(double balance) {this.
                      46   }
```

Fig. 15.9

`Account` class for storing records as objects.

Plain Old Java Objects

JAXB works with **POJOs (plain old Java objects)**—no special superclasses or interfaces are required for XML-serialization support. By default, JAXB serializes only an object's `public` instance variables and `public` *read–write* properties. Recall from Section 13.4.1 that a read–write property is defined by creating *get* and *set* methods with specific naming conventions. In class `Account`, methods `getAccountNumber` and `setAccountNumber` (lines 22–26) define a read–write property named `accountNumber`. Similarly, the *get* and *set* methods in lines 29–45 define the read–write properties `firstName`, `lastName` and `balance`. The class must also provide a public default or no-argument constructor to recreate the objects when they're read from the file.

Declaring Class `Accounts`

As you'll see in Fig. 15.11, this example stores `Account` objects in a `List<Account>`, then serializes the entire `List` into a file with one operation. To serialize a `List`, it must be defined as an instance variable of a class. For that

reason, we encapsulate the `List<Account>` in class `Accounts` (Fig. 15.10).

```
1   // Fig. 15.10: Accounts.java
2   // Maintains a List<Account>
3   import java.util.ArrayList;
4   import java.util.List;
5   import javax.xml.bind.annotation.XmlElement;
6
7   public class Accounts {
8      // @XmlElement specifies XML element name for
9      @XmlElement(name="account")
10     private List<Account> accounts = new ArrayLis
11
12     // returns the List<Accounts>
13     public List<Account> getAccounts() {return ac
14  }
```

Fig. 15.10

`Account` class for serializable objects.

Lines 9–10 declare and initialize the `List<Account>` instance variable `accounts`. JAXB enables you to customize many aspects of XML serialization, such as serializing a private instance variable or a read-only property. The annotation `@XMLElement` (line 9; package `javax.xml.bind.annotation`) indicates that the `private` instance variable should be serialized. We'll discuss the annotation's `name` argument shortly. The annotation is

required because the instance variable is not `public` and there's no corresponding `public` read–write property.

Writing XML Serialized Objects to a File

The program of Fig. 15.11 serializes an `Accounts` object to a text file. The program is similar to the one in Section 15.4, so we focus only on the new features. Line 9 imports the `JAXB` **class** from package `javax.xml.bind`. This package contains many related classes that implement the XML serializations we perform, but the `JAXB` class contains easy-to-use `static` methods that perform the most common operations.

```
   1   // Fig. 15.11: CreateSequentialFile.java
 2   // Writing objects to a file with JAXB and Buffe
        3   import java.io.BufferedWriter;
          4   import java.io.IOException;
          5   import java.nio.file.Files;
          6   import java.nio.file.Paths;
   7   import java.util.NoSuchElementException;
          8   import java.util.Scanner;
          9   import javax.xml.bind.JAXB;
                      10
     11   public class CreateSequentialFile {
 12      public static void main(String[] args) {
13          // open clients.xml, write objects to it t
      14          try(BufferedWriter output =
15             Files.newBufferedWriter(Paths.get("clie
                      16
17             Scanner input = new Scanner(System.in);
```

```
                    18
19            // stores the Accounts before XML seria
20            Accounts accounts = new Accounts();
                    21
  22            System.out.printf("%s%n%s%n? ",
23               "Enter account number, first name, l
24               "Enter end-of-file indicator to end
                    25
26            while (input.hasNext()) { // loop until
              27               try {
    28                 // create new record
29                Account record = new Account(inpu
30                   input.next(), input.next(), in
                    31
    32                 // add to AccountList
33                accounts.getAccounts().add(record
                34               }
35               catch (NoSuchElementException elemen
36                System.err.println("Invalid input
37                input.nextLine(); // discard inpu
                38               }
                    39
    40              System.out.print("? ");
                41            }
                    42
43            // write AccountList's XML to output
  44            JAXB.marshal(accounts, output);
                   45         }
   46         catch (IOException ioException) {
47            System.err.println("Error opening file.
                    48         }
                     49      }
                       50   }
```

```
Enter account number, first name, last name and balan
    Enter end-of-file indicator to end input.
              ? 100 Bob Blue 24.98
            ? 200 Steve Green -345.67
```

```
? 300 Pam White 0.00
? 400 Sam Red -42.16
? 500 Sue Yellow 224.62
? ^Z
```

Fig. 15.11

Writing objects to a file with `JAXB` and `BufferedWriter`.

To open the file, lines 14–15 call `Files` `static` method `newBufferedWriter`, which receives a `Path` specifying the file to open for writing (`"clients.xml"`) and—if the file exists—returns a `BufferedWriter` that class `JAXB` will use to write text to the file. Existing files that are opened for output in this manner are *truncated*. The standard filename extension for XML files is `.xml`. Lines 14–15 throw an `IOException` if a problem occurs while opening the file—such as, when the program does not have permission to access the file or when a read-only file is opened for writing. If so, the program displays an error message (lines 46–48), then terminates. Otherwise, variable `output` can be used to write to the file.

Line 20 creates the `Accounts` object that contains the `List<Account>`. Lines 26–41 input each record, create an `Account` object (lines 29–30) and add to the `List` (line 33).

When the user enters the end-of-file indicator to terminate input, line 44 uses class `JAXB`'s `static` method `marshal`

to serialize as XML the `Accounts` object containing the `List<Account>`. The first argument is the object to serialize. The second argument to this particular overload of method `marshal` is a `Writer` (package `java.io`) that's used to output the XML—`BufferedWriter` is a subclass of `Writer`. The `BufferedWriter` obtained in lines 14–15 outputs the XML to a file.

Note that only one statement is required to write the *entire* `Accounts` object and all of the objects in its `List<Account>`. In the sample execution for the program in Fig. 15.11, we entered information for five accounts—the same information shown in Fig. 15.5.

The XML Output

Figure 15.12 shows the contents of the file `clients.xml`. Though you do not need to know XML to work with this example, note that the XML is human readable. When JAXB serializes an object of a class, it uses the class's name with a lowercase first letter as the corresponding XML element name, so the `accounts` element (lines 2–33) represents the `Accounts` object.

Recall that line 9 in class `Accounts` (Fig. 15.10) preceded the `List<Account>` instance variable with the annotation

```
@XmlElement(name="account")
```

In addition to enabling JAXB to serialize the instance variable, this annotation specifies the XML element name (`"account"`) used to represent each of the `List`'s `Account` objects in the serialized output. For example, lines 3–8 in Fig. 15.12 represent the `Account` for `Bob Blue`. If we did not specify the annotation's `name` argument, the instance variable's name (`accounts`) would have been used as the XML element name. Many other aspects of JAXB XML serialization are customizable. For more details, see

```
https://docs.oracle.com/javase/tutorial/jaxb/intro/
```

```
 1  <?xml version="1.0" encoding="UTF-8" standalone=
 2     <accounts>
 3        <account>
 4           <accountNumber>100</accountNumber>
 5           <balance>24.98</balance>
 6           <firstName>Bob</firstName>
 7           <lastName>Blue</lastName>
 8        </account>
 9        <account>
10           <accountNumber>200</accountNumber>
11           <balance>-345.67</balance>
12           <firstName>Steve</firstName>
13           <lastName>Green</lastName>
14        </account>
15        <account>
16           <accountNumber>300</accountNumber>
17           <balance>0.0</balance>
18           <firstName>Pam</firstName>
19           <lastName>White</lastName>
20        </account>
21        <account>
22           <accountNumber>400</accountNumber>
```

```
23            <balance>-42.16</balance>
24            <firstName>Sam</firstName>
25            <lastName>Red</lastName>
26        </account>
27        <account>
28            <accountNumber>500</accountNumber>
29            <balance>224.62</balance>
30            <firstName>Sue</firstName>
31            <lastName>Yellow</lastName>
32        </account>
33    </accounts>
```

Fig. 15.12

Contents of `clients.xml`.

Each of class `Account`'s property has a corresponding XML element with the same name as the property. For example, lines 4–7 are the XML elements for `Bob Blue`'s `accountNumber`, `balance`, `firstName` and `lastName`—JAXB placed the XML elements in alphabetical order, though this is not required or guaranteed. Within each of these elements is the corresponding property's value—`100` for the `accountNumber`, `24.98` for the balance, `Bob` for the `firstName` and `Blue` for the `lastName`. Lines 9–32 represent the other four `Account` objects that we input into the program in the sample execution.

15.5.2 Reading and

Deserializing Data from a Sequential File

The preceding section showed how to create a file containing XML serialized objects. In this section, we discuss how to *read serialized data* from a file. Figure 15.13 reads objects from the file created by the program in Section 15.5.1, then displays the contents. The program opens the file for input by calling `Files` `static` method `newBufferedReader`, which receives a `Path` specifying the file to open and, if the file exists and no exceptions occur, returns a `BufferedReader` for reading from the file.

```
   1    // Fig. 15.13: ReadSequentialFile.java
 2    // Reading a file of XML serialized objects wit
 3    // BufferedReader and displaying each object.
       4    import java.io.BufferedReader;
         5    import java.io.IOException;
          6    import java.nio.file.Files;
          7    import java.nio.file.Paths;
          8    import javax.xml.bind.JAXB;
                          9
     10    public class ReadSequentialFile {
 11       public static void main(String[] args) {
 12          // try to open file for deserialization
      13          try(BufferedReader input =
 14             Files.newBufferedReader(Paths.get("cli
  15             // unmarshal the file's contents
 16             Accounts accounts = JAXB.unmarshal(inp
                        17
        18             // display contents
 19             System.out.printf("%-10s%-12s%-12s%10s
 20                "First Name", "Last Name", "Balance
                         21
```

```
22             for (Account account : accounts.getAcc
23                System.out.printf("%-10d%-12s%-12s%
24                   account.getAccountNumber(), acco
25                   account.getLastName(), account.g
                  26             }
                   27          }
   28           catch (IOException ioException) {
29              System.err.println("Error opening file
                   30          }
                    31       }
                     32    }
```

Account	First Name	Last Name	Balance
100	Bob	Blue	24.98
200	Steve	Green	-345.67
300	Pam	White	0.00
400	Sam	Red	-42.16
500	Sue	Yellow	224.62

```
No more records
```

Fig. 15.13

Reading a file of XML serialized objects with `JAXB` and a `BufferedReader` and displaying each object.

Line 16 uses `JAXB` `static` method `unmarshal` to read the

contents of `clients.xml` and convert the XML into an `Accounts` object. The overload of `unmarshal` used here reads XML from a `Reader` (package `java.io`) and creates an object of the type specified as the second argument —`BufferedReader` is a subclass of `Reader`. The `BufferedReader` obtained in lines 13–14 reads text from a file. Method `unmarshal`'s second argument is a `Class<T>` object (package `java.lang`) representing the type of the object to create from the XML—the notation `Accounts.class` is a Java compiler shorthand for

```
new Class<Accounts>
```

Once again, note that one statement reads the entire file and recreates the `Accounts` object. If no exceptions occur, lines 19–26 display the contents of the `Accounts` object.