9.3.2 Case Study: Object Serialization

In all the applications seen so far, all objects were created and used as the program was running. But there are situations in which objects may need to be created in one run and used in another. Here are some scenarios:

- After creating a collection and adding elements to it, the user may want to quit the program for the day and come back the next day to resume working. The user expects that all elements will remain available on the next day, and in the same state.
- After creating an object through a program, the user may want to be able to launch another program to process the object created in the first; that is, the second app must be able to access objects created by the first.
- After creating an object through a program, the user may want to be able to share it with someone else; that is, make the created object accessible to a remote location.

All these scenarios can be realized if we can somehow "save" objects in some persistent storage, one that does not disappear when our program ends. This means we need to write code that writes all the attributes of the object to a disk file. Primitive attributes can easily be written, but non-primitive attributes need to be handled recursively. This is because an object attribute of a class refers to another class and that class may itself have object attributes, and so on. This recursion will ultimately yield primitive or string data. This complex process is called **serializing** the object, and it culminates in a set of binary data that captures the state of the object. Given this data, and the classes of which the object and its attributes are instances, we should be able to reconstitute the object. This reverse process is called deserialization.

Java comes with ready-made components that implement both processes. The class

```
java.io.ObjectOutputStream
```

is responsible for serialization. Its constructor takes an `OutputStream` as a sink (a place to which the data should be written). Hence, if we want to serialize an object to a disk file, we provide a file as a sink:

```
FileOutputStream fos = new
    FileOutputStream(filename);
ObjectOutputStream oos = new
    ObjectOutputStream(fos);
```

The key method in the class is

```
void writeObject(Object obj)
```

Thanks to the object hierarchy and the substitutability principle, this method accepts *any* object type parameter. As an example, given an instance `gc` of `GlobalCredit`, a collection of `CreditCard` objects, we can serialize it as follows:

```
oos.writeObject(gc);
oos.close();
```

Note that `gc` may contain thousands of credit cards and that each credit card aggregates two dates and a host of attributes, yet this simple invocation will store everything in the file. The process can be reversed with the help of the class:

```
java.io.ObjectInputStream
```

Here is a fragment that deserializes the collection:

```
FileInputStream fis = new
    FileInputStream(filename);
ObjectInputStream ois = new
    ObjectInputStream(fis);
gc = (GlobalCredit)
    ois.readObject();
ois.close();
```

Note that the `readObject` method returns an `Object`; hence, the return must be cast to the proper type before the application can use the deserialized object. This is a weak point in the process because the cast may lead to a runtime error when the actual type of the serialized object is different from, and not a subclass of, the cast type. It is therefore prudent to check the type before casting:

```
if (ois.readObject() instanceof
    GlobalCredit)
```

The object stream methods throw several types of exceptions, and until we learn how to handle them (Chapter 11), we must add a suffix to our `main` method:

```
public static void main(String[]
    args) throws Exception
```