20.5 Object Serialization

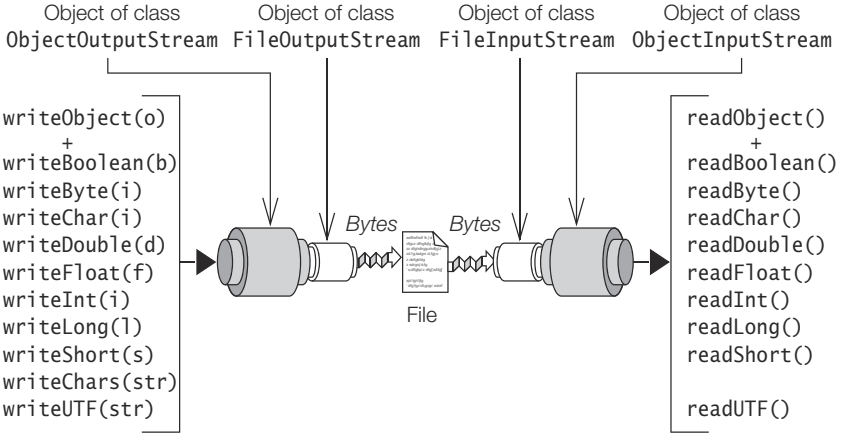
Object serialization allows the state of an object to be transformed into a sequence of bytes that can be converted back into a copy of the object (called *deserialization*). After deserialization, the object has the same state as it had when it was serialized, barring any data members that were not serializable. This mechanism is generally known as *persistence*—the serialized result of an object can be stored in a repository from which it can be later retrieved.

Java provides the object serialization facility through the `ObjectInput` and `ObjectOutput` interfaces, which allow the writing and reading of objects to and from I/O streams. These two interfaces extend the `DataInput` and `DataOutput` interfaces, respectively (Figure 20.1, p. 1235).

The `ObjectOutputStream` class and the `ObjectInputStream` class implement the `ObjectOutput` interface and the `ObjectInput` interface, respectively, providing methods to write and read binary representation of both objects as well as Java primitive values. Figure 20.9 gives an overview of how these classes can be chained to underlying streams and some selected methods they provide. The figure does not show the methods inherited from the abstract `OutputStream` and `InputStream` superclasses.

The read and write methods in the two classes can throw an `IOException`, and the read methods will throw an `EOFException` if an attempt is made to read past the end of the stream.

Figure 20.9 *Object Stream Chaining*



The `ObjectOutputStream` Class

The class `ObjectOutputStream` can write objects to any stream that is a subclass of the `OutputStream`—for example, to a file or a network connection (socket). An `ObjectOutputStream` must be chained to an `OutputStream` using the following constructor:

```
ObjectOutputStream(OutputStream out) throws IOException
```

For example, in order to store objects in a file and thus provide persistent storage for objects, an `ObjectOutputStream` can be chained to a `FileOutputStream`:

```
FileOutputStream outputFile = new FileOutputStream("obj-storage.dat");
ObjectOutputStream outputStream = new ObjectOutputStream(outputFile);
```

Objects can be written to the stream using the `writeObject()` method of the `ObjectOutputStream` class:

```
final void writeObject(Object obj) throws IOException
```

The `writeObject()` method can be used to write *any* object to a stream, including strings and arrays, as long as the object implements the `java.io.Serializable` interface, which is a *marker interface* with no methods. The `String` class, the primitive wrapper classes, and all array types implement the `Serializable` interface. A serializable object can be any compound object containing references to other objects, and all constituent objects that are serializable are serialized recursively when the compound object is written out. This is true even if there are cyclic references between the objects. Each object is written out only once during serialization. The following information is included when an object is serialized:

- The class information needed to reconstruct the object
- The values of all serializable non-transient and non-static members, including those that are inherited

A checked exception of the type `java.io.NotSerializableException` is thrown if a non-serializable object is encountered during the serialization process. Note also that objects of subclasses that extend a serializable class are always serializable.

The `ObjectInputStream` Class

An `ObjectInputStream` is used to restore (*deserialize*) objects that have previously been serialized using an `ObjectOutputStream`. An `ObjectInputStream` must be chained to an `InputStream`, using the following constructor:

```
ObjectInputStream(InputStream in) throws IOException
```

For example, in order to restore objects from a file, an `ObjectInputStream` can be chained to a `FileInputStream`:

```
FileInputStream inputFile = new FileInputStream("obj-storage.dat");
ObjectInputStream inputStream = new ObjectInputStream(inputFile);
```

The `readObject()` method of the `ObjectInputStream` class is used to read the serialized state of an object from the stream:

```
final Object readObject() throws ClassNotFoundException, IOException
```

Note that the reference type of the returned object is `Object`, regardless of the actual type of the retrieved object, and can be cast to the desired type. Objects and values must be read in the same order as when they were serialized.

Serializable, non-transient data members of an object, including those data members that are inherited, are restored to the values they had at the time of serialization. For compound objects containing references to other objects, the constituent objects are read to re-create the whole object structure. In order to deserialize objects, the appropriate classes must be available at runtime. Note that new objects are created during deserialization, so that no existing objects are overwritten.

The class `ObjectSerializationDemo` in Example 20.6 serializes some objects in the `writeData()` method at (1), and then deserializes them in the `readData()` method at (2). The `readData()` method also writes the data to the standard output stream.

The `writeData()` method at (1) writes the following values to the output stream: an array of strings (`strArray`), a `long` value (`num`), an array of `int` values (`intArray`), a `String` object (`commonStr`) which is shared with the array `strArray` of strings, and an instance (`oneCD`) of the record class `CD` whose component fields are all serializable.

Duplication is automatically avoided when the same object is serialized several times. The shared `String` object (`commonStr`) is actually only serialized once. Note that the array elements and the characters in a `String` object are not written out explicitly one by one. It is enough to pass the object reference in the `writeObject()` method call. The method also recursively goes through the array of strings, `strArray`, serializing each `String` object in the array. The current state of the `oneCD` instance is also serialized.

The method `readData()` at (2) deserializes the data in the order in which it was written. An explicit cast is needed to convert the reference of a deserialized object to a subtype. Applying the right cast is of course the responsibility of the application. Note that new objects are created by the `readObject()` method, and that an object created during the deserialization process has the same state as the object that was serialized.

Efficient Record Serialization

Example 20.6 shows an example of serializing and deserializing an instance of a record class (§5.14, p. 299). It is worth noting that both processes on records are very efficient as the state components of a record entirely describe the state values to serialize, and as the canonical constructor of a record class is always used to create the complete state of a record, the canonical constructor is also always used during deserialization. By design, selective and customized serialization, discussed later in this section, are not allowed for records.

Example 20.6 *Object Serialization*

```
import java.io.Serializable;
import java.time.Year;
/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,
                 Year year, Genre genre) implements Serializable {
```

```
  public enum Genre implements Serializable {POP, JAZZ, OTHER}
}
```

```
//Reading and Writing Objects
import java.io.*;
import java.time.Year;
import java.util.Arrays;

public class ObjectSerializationDemo {
  void writeData() {                                    // (1)
    try (// Set up the output stream:
        FileOutputStream outputFile = new FileOutputStream("obj-storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {

      // Write data:
      String[] strArray = {"Seven", "Eight", "Six"};
      long num = 2014;
      int[] intArray = {1, 3, 1949};
      String commonStr = strArray[2];                   // "Six"
      CD oneCD = new CD("Jaav", "Java Jive", 8, Year.of(2017), CD.Genre.POP);
      outputStream.writeObject(strArray);
      outputStream.writeLong(num);
      outputStream.writeObject(intArray);
      outputStream.writeObject(commonStr);
      outputStream.writeObject(oneCD);

    } catch (FileNotFoundException e) {
      System.err.println("File not found: " + e);
    } catch (IOException e) {
      System.err.println("Write error: " + e);
    }
  }

  void readData() {                                     // (2)
    try (// Set up the input stream:
        FileInputStream inputFile = new FileInputStream("obj-storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read the data:
      String[] strArray = (String[]) inputStream.readObject();
      long num = inputStream.readLong();
      int[] intArray = (int[]) inputStream.readObject();
      String commonStr = (String) inputStream.readObject();
      CD oneCD = (CD) inputStream.readObject();

      // Write data to the standard output stream:
      System.out.println(Arrays.toString(strArray));
      System.out.println(num);
      System.out.println(Arrays.toString(intArray));
      System.out.println(commonStr);
      System.out.println(oneCD);

    } catch (FileNotFoundException e) {
      System.err.println("File not found: " + e);
```

```
      } catch (EOFException e) {
        System.err.println("End of stream: " + e);
      } catch (IOException e) {
        System.err.println("Read error: " + e);
      } catch (ClassNotFoundException e) {
        System.err.println("Class not found: " + e);
      }
    }

    public static void main(String[] args) {
      ObjectSerializationDemo demo = new ObjectSerializationDemo();
      demo.writeData();
      demo.readData();
    }
}
```

Output from the program:

```
[Seven, Eight, Six]
2014
[1, 3, 1949]
Six
CD[artist=Jaav, title=Java Jive, noOfTracks=8, year=2017, genre=POP]
```

Selective Serialization

As noted earlier, static fields are not serialized, as these are not part of the state of an object. An instance field of an object can be omitted from being serialized by specifying the `transient` modifier in the declaration of the field—typically used for sensitive data in a field. Selective serialization discussed here is *not* applicable to record classes.

Example 20.7 illustrates some salient aspects of serialization. The setup comprises the classes `Wheel` and `Unicycle`, and their client class `SerialClient`. The class `Unicycle` has a field of type `Wheel`, and the class `Wheel` has a field of type `int`. The class `Unicycle` is a compound object with a `Wheel` object as a constituent object. The class `SerialClient` serializes and deserializes a unicycle in the `try`-with-resources statements at (4) and (5), respectively. The state of the objects is printed to the standard output stream before serialization, and so is the state of the object created by deserialization.

Both the Compound Object and Its Constituents Are Serializable

If we run the program with the following declarations for the `Wheel` and the `Unicycle` classes, where a compound object of the serializable class `Unicycle` uses an object of the serializable class `Wheel` as a constituent object:

```
class Wheel implements Serializable {                          // (1a)
  private int wheelSize;
  ...
}
```

```
class Unicycle implements Serializable {                          // (2)
  private Wheel wheel;                                            // (3a)
  ...
}
```

we get the following output, showing that both serialization and deserialization were successful:

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with wheel size: 65
```

A compound object with its constituent objects is often referred to as an *object graph*. Serializing a compound object serializes its complete object graph—that is, the compound object and its constituent objects are recursively serialized.

Example 20.7 *Non-Serializable Objects*

```
import java.io.Serializable;

// public class Wheel implements Serializable {                  // (1a)
public class Wheel {                                              // (1b)
  private int wheelSize;

  public Wheel(int ws) { wheelSize = ws; }

  @Override
  public String toString() { return "wheel size: " + wheelSize; }
}
```

```
import java.io.Serializable;

public class Unicycle implements Serializable {                   // (2)
  private Wheel wheel;                                            // (3a)
//transient private Wheel wheel;                                  // (3b)

  public Unicycle (Wheel wheel) { this.wheel = wheel; }

  @Override
  public String toString() { return "Unicycle with " + wheel; }
}
```

```
import java.io.*;

public class SerialClient {

  public static void main(String args[])
      throws IOException, ClassNotFoundException {

    try (// Set up the output stream:                             // (4)
        FileOutputStream outputFile = new FileOutputStream("storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {
```

```
        // Write the data:
        Wheel wheel = new Wheel(65);
        Unicycle uc = new Unicycle(wheel);
        System.out.println("Before writing: " + uc);
        outputStream.writeObject(uc);
      }

      try (// Set up the input streams:                              // (5)
          FileInputStream inputFile = new FileInputStream("storage.dat");
          ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

        // Read data.
        Unicycle uc = (Unicycle) inputStream.readObject();

        // Write data on standard output stream.
        System.out.println("After reading: " + uc);
      }
    }
}
```

Transient Fields Are Not Serializable

If we declare the `wheel` field of the `Unicycle` class in Example 20.7 to be `transient` for the sake of demonstrating how such fields are handled at serialization, (3b):

```
class Wheel implements Serializable {                               // (1a)
  private int wheelSize;
  ...
}

class Unicycle implements Serializable {                            // (2)
  transient private Wheel wheel;                                    // (3b)
  ...
}
```

we get the following output, showing that the `wheel` field of the `Unicycle` object was not serialized:

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with null
```

Serializable Compound Object with Non-Serializable Constituents

In order for a compound object to be serialized, its constituent objects must be serializable; otherwise, serialization will not succeed. If the class `Wheel` in Example 20.7 is *not* serializable, (1b):

```
class Wheel {                                                       // (1b)
  private int wheelSize;
  ...
}
```

```
class Unicycle implements Serializable {                          // (2)
  private Wheel wheel;                                            // (3a)
  ...
}
```

we get the following output when we run the program—that is, a `Unicycle` object *cannot* be serialized because its constituent `Wheel` object is not serializable:

```
>java SerialClient
Before writing: Unicycle with wheel size: 65
Exception in thread "main" java.io.NotSerializableException: Wheel
  ...
  at SerialClient.main(SerialClient.java:20)
```

Customizing Object Serialization

As we have seen, the class of the object must implement the `Serializable` interface if we want the object to be serialized. If this object is a compound object, then all its constituent objects must also be serializable, and so on.

It is not always possible for a client to declare that a class is `Serializable`. It might be declared `final`, and therefore not extendable. The client might not have access to the code, or extending this class with a serializable subclass might not be an option. Java provides a customizable solution for serializing objects in such cases.

Customized serialization discussed here is *not* applicable to record classes.

The basic idea behind the scheme is to use default serialization as much as possible, and to provide *hooks* in the code for the serialization mechanism to call specific methods to deal with objects or values that should not or cannot be serialized by the default methods of the object streams.

Customizing serialization is illustrated in Example 20.8, using the `Wheel` and `Unicycle` classes from Example 20.7. The serializable class `Unicycle` would like to use the `Wheel` class, but this class is not serializable. If the `wheel` field in the `Unicycle` class is declared to be `transient`, it will be ignored by the default serialization procedure. This is not a viable option, as the unicycle will be missing the wheel size when a serialized unicycle is deserialized, as was illustrated in Example 20.7.

Any serializable object has the option of customizing its own serialization if it implements the following pair of methods:

```
private void writeObject(ObjectOutputStream) throws IOException;
private void readObject(ObjectInputStream)
                    throws IOException, ClassNotFoundException;
```

These methods are *not* part of any interface. Although private, these methods can be called by the JVM. The first method above is called on the object when its serialization starts. The serialization procedure uses the reference value of the object to be serialized that is passed in the call to the `ObjectOutputStream.writeObject()` method, which in turn calls the first method above on this object. The second method above is called on the object created when the deserialization procedure is initiated by the call to the `ObjectInputStream.readObject()` method.

Customizing serialization for objects of the class `Unicycle` in Example 20.8 is achieved by the private methods at (3c) and (3d). Note that the field `wheel` is declared `transient` at (3b) and excluded by the normal serialization process.

In the private method `writeObject()` at (3c) in Example 20.8, the pertinent lines of code are the following:

```
oos.defaultWriteObject();               // Method in the ObjectOutputStream class
oos.writeInt(wheel.getWheelSize());     // Method in the ObjectOutputStream class
```

The call to the `defaultWriteObject()` method of the `ObjectOutputStream` does what its name implies: normal serialization of the current object. The second line of code does the customization: It writes the binary `int` value of the wheel size to the `ObjectOutputStream`. The code for customization can be called both before and after the call to the `defaultWriteObject()` method, as long as the same order is used during deserialization.

In the private method `readObject()` at (3d), the pertinent lines of code are the following:

```
ois.defaultReadObject();                // Method in the ObjectInputStream class
int wheelSize = ois.readInt();          // Method in the ObjectInputStream class
this.wheel = new Wheel(wheelSize);
```

The call to the `defaultReadObject()` method of the `ObjectInputStream` does what its name implies: normal deserialization of the current object. The second line of code reads the binary `int` value of the wheel size from the `ObjectInputStream`. The third line of code creates a `Wheel` object, passes this value in the constructor call, and assigns its reference value to the `wheel` field of the current object. Again, code for customization can be called both before and after the call to the `defaultReadObject()` method, as long as it is in correspondence with the customization code in the `writeObject()` method.

The client class `SerialClient` in Example 20.8 is the same as the one in Example 20.7. The output from the program confirms that the object state prior to serialization is identical to the object state after deserialization.

Example 20.8 *Customized Serialization*

```
public class Wheel {                                                    // (1b)
  private int wheelSize;

  public Wheel(int ws) { wheelSize = ws; }

  public int getWheelSize() { return wheelSize; }

  @Override
  public String toString() { return "wheel size: " + wheelSize; }
}
```

```
import java.io.*;

public class Unicycle implements Serializable {                    // (2)
  transient private Wheel wheel;                                   // (3b)

  public Unicycle(Wheel wheel) { this.wheel = wheel; }

  @Override
  public String toString() { return "Unicycle with " + wheel; }

  private void writeObject(ObjectOutputStream oos) {               // (3c)
    try {
      oos.defaultWriteObject();
      oos.writeInt(wheel.getWheelSize());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  private void readObject(ObjectInputStream ois) {                 // (3d)
    try {
      ois.defaultReadObject();
      int wheelSize = ois.readInt();
      this.wheel = new Wheel(wheelSize);
    } catch (IOException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

```
public class SerialClient { // Same as in Example 20.7 }
```

Output from the program:

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with wheel size: 65
```

Serialization and Inheritance

The inheritance hierarchy of an object also determines what its state will be after it is deserialized. An object will have the same state at deserialization as it had at the time it was serialized if *all* its superclasses are also serializable. This is because the normal object creation and initialization procedure using constructors is *not* run during deserialization.

However, if any superclass of an object is *not* serializable, then the normal creation procedure using *no-argument* or *default* constructors *is* run, starting at the first non-serializable superclass, all the way up to the `Object` class. This means that the state at deserialization might not be the same as at the time the object was serialized

because superconstructors run during deserialization may have initialized the state of the object. If the non-serializable superclass does not provide a non-argument constructor or the default constructor, a `java.io.InvalidClassException` is thrown during deserialization.

Example 20.9 illustrates how inheritance affects serialization. The `Student` class is a subclass of the `Person` class. Whether the superclass `Person` is serializable or not has implications for serializing objects of the `Student` subclass, in particular, when their byte representation is deserialized.

Superclass Is Serializable

If the superclass is serializable, then any object of a subclass is also serializable. In Example 20.9, the code at (4) in the class `SerialInheritance` serializes a `Student` object:

```
Student student = new Student("Pendu", 1007);
System.out.println("Before writing: " + student);
outputStream.writeObject(student);
```

The corresponding code for deserialization of the streamed object is at (5) in the class `SerialInheritance`:

```
Student student = (Student) inputStream.readObject();
System.out.println("After reading: " + student);
```

We get the following output from the program in Example 20.9 when it is run with (1a) and (3a) in the `Person` class and the `Student` class, respectively—that is, when the superclass is serializable and so is the subclass, by virtue of inheritance.

The results show that the object state prior to serialization is identical to the object state after deserialization. In this case, no superclass constructors were run during deserialization.

```
Before writing: Student state(Pendu, 1007)
After reading: Student state(Pendu, 1007)
```

Superclass Is Not Serializable

However, the result of deserialization is not the same when the superclass `Person` is not serializable, but the subclass `Student` is. We get the following output from the program in Example 20.9 when it is run with (1b) and (3b) in the `Person` class and the `Student` class, respectively—that is, when only the subclass is serializable, but not the superclass. The output shows that the object state prior to serialization is not identical to the object state after deserialization.

```
Before writing: Student state(Pendu, 1007)
No-argument constructor executed.
After reading: Student state(null, 1007)
```

During deserialization, the *zero-argument* constructor of the `Person` superclass at (2) is run. As we can see from the declaration of the `Person` class in Example 20.9, this zero-argument constructor does not initialize the `name` field, which remains initialized with the default value for reference types (`null`).

If the superclass `Person` does not provide the no-argument constructor or the default constructor, as in the declaration below, the call to the `readObject()` method to perform deserialization throws an `InvalidClassException`.

```
public class Person  {                                          // (1b)
  private String name;

  public Person(String name) { this.name = name; }

  public String getName() { return name; }
}
```

Output from the program (*edited to fit on the page*):

```
Before writing: Student state(Pendu, 1007)
Exception in thread "main" java.io.InvalidClassException:
        Student; no valid constructor
        ...
        at SerialInheritance.main(SerialInheritance.java:28)
```

The upshot of serializing objects of subclasses is that the superclass should be serializable, unless there are compelling reasons for why it is not. And if the superclass is not serializable, it should at least provide either the default constructor or the no-argument constructor to avoid an exception during deserialization.

Although a superclass might be serializable, its subclasses can prevent their objects from being serialized by implementing the `private` method `writeObject(ObjectOutputStream)` that throws a `java.io.NotSerializableException`.

Example 20.9 *Serialization and Inheritance*

```
import java.io.Serializable;

// A superclass
public class Person implements Serializable {                  // (1a)
//public class Person  {                                        // (1b)
  private String name;

  public Person() {                                             // (2)
    System.out.println("No-argument constructor executed.");
  }
  public Person(String name) { this.name = name; }

  public String getName() { return name; }
}
```

```
import java.io.Serializable;

public class Student extends Person {                           // (3a)
//public class Student extends Person implements Serializable {  // (3b)

  private long studNum;

  public Student(String name, long studNum) {
    super(name);
```

```
    this.studNum = studNum;
  }

  @Override
  public String toString() {
    return "Student state(" + getName() + ", " + studNum + ")";
  }
}
```

```
import java.io.*;

public class SerialInheritance {
  public static void main(String[] args)
      throws IOException, ClassNotFoundException {

    // Serialization:
    try (// Set up the output stream:                                          (4)
        FileOutputStream outputFile = new FileOutputStream("storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {

      // Write data:
      Student student = new Student("Pendu", 1007);
      System.out.println("Before writing: " + student);
      outputStream.writeObject(student);
    }

    // Deserialization:
    try (// Set up the input stream:                                           (5)
        FileInputStream inputFile = new FileInputStream("storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read data.
      Student student = (Student) inputStream.readObject();

      // Write data on standard output stream.
      System.out.println("After reading: " + student);
    }
  }
}
```

Serialization and Versioning

Class versioning comes into play when we serialize an object with one definition of a class, but deserialize the streamed object with a different class definition. By streamed object, we mean the serialized representation of an object. Between serialization and deserialization of an object, the class definition can change.

Note that at serialization and at deserialization, the definition of the class (i.e., its bytecode file) should be accessible. In the examples so far, the class definition has been the same at both serialization and deserialization. Example 20.10 illustrates the problem of class definition mismatch at deserialization and the solution provided by Java.

Example 20.10 makes use of the following classes (numbering refers to code lines in the example):

(1) The original version of the serializable class `Item`. It has one field named `price`. An object of this class will be serialized and read using different versions of this class.

(2) A newer version of the class `Item` that has been augmented with a field for the weight of an item. This class will only be used for deserialization of objects that have been serialized with the original version of the class.

(3) The class `Serializer` serializes an object of the original version of the class `Item`.

(4) The class `DeSerializer` deserializes a streamed object of the class `Item`. In the example, deserialization is based on a different version of the `Item` class than the one used at serialization.

There are no surprises if we use the original class `Item` to serialize and deserialize an object of the `Item` class.

```
// Original version of the Item class.
public class Item implements Serializable {                          // (1)
  private double price;
//...
}
```

Result:

```
Before writing: Price: 100.00
After reading: Price: 100.00
```

If we deserialize a streamed object of the original class `Item` at (1) based on the byte-code file of the augmented version of the `Item` class at (2), an `InvalidClassException` is thrown at runtime.

```
// New version of the Item class.
public class Item implements Serializable {                          // (2)
  private double price;
  private double weight;                                  // Additional field
//...
}
```

Result (*edited to fit on the page*):

```
Exception in thread "main" java.io.InvalidClassException: Item;
local class incompatible:
  stream classdesc serialVersionUID = -4194294879924868414,
  local class serialVersionUID = -1186964199368835959
     ...
     at DeSerializer.main(DeSerializer.java:14)
```

The question is, how was the class definition mismatch discovered at runtime? The answer lies in the stack trace of the exception thrown. The local class was incompatible, meaning the class we are using to deserialize is not compatible with the class that was used when the object was serialized. In addition, two long numbers are

printed, representing the `serialVersionUID` of the respective class definitions. The first `serialVersionUID` is generated by the serialization process based on the class definition and becomes part of the streamed object. The second `serialVersionUID` is generated based on the local class definition that is accessible at deserialization. The two are not equal, and deserialization fails.

A serializable class can provide its `serialVersionUID` by declaring it in its class declaration, exactly as shown below, except for the initial value which of course can be different:

```
static final long serialVersionUID = 100L;            // Appropriate value.
```

As we saw in the example above, if a serializable class does not provide a `serialVersionUID`, one is implicitly generated. By providing an explicit `serialVersionUID`, it is possible to control what happens at deserialization. As newer versions of the class are created, the `serialVersionUID` can be kept the same until it is deemed that older streamed objects are no longer compatible for deserialization. After the change to the `serialVersionUID`, it will not be possible to deserialize older streamed objects of the class based on newer versions of the class. Although static members of a class are not serialized, the only exception is the value of the `static final long serialVersionUID` field.

In the scenario below, the original version and the newer version of the class `Item` both declare a `serialVersionUID` at (1a) and at (2a), respectively, that has the same value. An `Item` object is serialized using the original version, but deserialized based on the newer version. We see that serialization succeeds, and the `weight` field is initialized to the default value `0.0`. In other words, the object created is of the newer version of the class.

```
// Original version of the Item class.
public class Item implements Serializable {    // (1)
  static final long serialVersionUID = 1000L;  // (1a)
  private double price;
//...
}

// New version of the Item class.
public class Item implements Serializable {    // (2)
  static final long serialVersionUID = 1000L;  // (2a) Same serialVersionUID
  private double price;
  private double weight;                       // Additional field
//...
}
```

Result:

```
Before writing: Price: 100.00
After reading: Price: 100.00, Weight: 0.00
```

However, if we now deserialize the streamed object of the original class having `1000L` as the `serialVersionUID`, based on the newer version of the class having the `serialVersionUID` equal to `1001L`, deserialization fails as we would expect because the `serialVersionUID`s are different.

```
// New version of the Item class.
public class Item implements Serializable {    // (2)
  static final long serialVersionUID = 1001L;  // (2b) Different serialVersionUID
  private double price;
  private double weight;
//...
}
```

Result (*edited to fit on the page*):

```
Exception in thread "main" java.io.InvalidClassException: Item;
local class incompatible:
  stream classdesc serialVersionUID = 1000,
  local class serialVersionUID = 1001
     ...
     at DeSerializer.main(DeSerializer.java:14)
```

Best practices advocate that serializable classes should use the `serialVersionUID` solution for better control of what happens at deserialization as classes evolve.

Example 20.10 *Class Versioning*

```
import java.io.Serializable;

// Original version of the Item class.
public class Item implements Serializable {                          // (1)
//static final long serialVersionUID = 1000L;                        // (1a)

  private double price;

  public Item(double price) {
    this.price = price;
  }

  @Override
  public String toString() {
    return String.format("Price: %.2f%n", this.price);
  }
}
```

```
import java.io.Serializable;

// New version of the Item class.
public class Item implements Serializable {                          // (2)
//static final long serialVersionUID = 1000L;                        // (2a)
//static final long serialVersionUID = 1001L;                        // (2b)

  private double price;
  private double weight;
  public Item(double price, double weight) {
    this.price = price;
    this.weight = weight;
  }
```

```
  @Override
  public String toString() {
    return String.format("Price: %.2f, Weight: %.2f", this.price, this.weight);
  }
}
```

```
// Serializer for objects of class Item.
import java.io.*;

public class Serializer {                                                    // (3)
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (// Set up the output stream:
        FileOutputStream outputFile = new FileOutputStream("item_storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {

      // Serialize an object of the original class:
      Item item = new Item(100.00);
      System.out.println("Before writing: " + item);
      outputStream.writeObject(item);
    }
  }
}
```

```
// Deserializer for objects of class Item.
import java.io.*;

public class DeSerializer{                                                   // (4)
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (// Set up the input streams:
        FileInputStream inputFile = new FileInputStream("item_storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read a serialized object of the Item class.
      Item item = (Item) inputStream.readObject();

      // Write data on standard output stream.
      System.out.println("After reading: " + item);
    }
  }
}
```

Review Questions

20.5 Which of the following best describes the data written by an `ObjectOutputStream`? Select the one correct answer.

(a) Bytes and other Java primitive types
(b) Object graphs