Streams and File I/O

Majid Ghaderi

Objectives

- Describe the concept of an I/O stream
- Explain the difference between text and binary files
- Save data, including objects, in a file
- Read data, including objects, from a file

Streams

- A *stream* is an object that enables the flow of data between a program and some I/O device or file
  - If the data flows into a program, then the stream is called an *input stream*
  - If the data flows out of a program, then the stream is called an *output stream*

Streams

- Input streams can flow from the keyboard or from a file
  - `System.in` is an input stream that connects to the keyboard

```
Scanner keyboard = new Scanner(System.in);
```

- Output streams can flow to a screen or to a file
  - `System.out` is an output stream that connects to the screen

```
System.out.println("Output stream");
```

Text Files and Binary Files

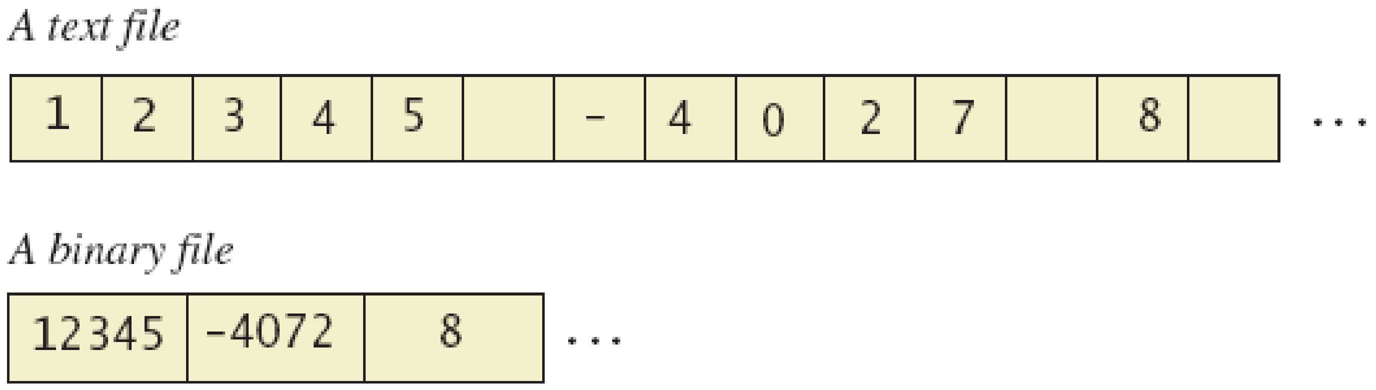
- Files that are designed to be read by human beings, and that can be read or written with an editor are called *text files*
  - Text files can also be called ASCII files because the data they contain uses an ASCII encoding scheme
  - An advantage of text files is that they are usually the same on all computers, so that they can move from one computer to another

Text Files and Binary Files

- Files that are designed to be read by programs and that consist of a sequence of binary digits are called *binary files*
  - Binary files are designed to be read on the same type of computer and with the same programming language as the computer that created the file
  - An advantage of binary files is that they are *more efficient to process* than text files
  - Unlike most binary files, Java binary files have the advantage of being platform independent also

Text Files and Binary Files

- A text file and a binary file containing the same values



Byte Streams and Character Streams

- Java IO system is quite large
  - There are many classes in the `java.io` package
  - Do not be intimidated!
- At the most basic level: Java defines two types of IO streams:
  - Byte streams: for handling input and output of bytes
  - Character streams: for handling input and output of Unicode (multi-byte) characters

Byte Stream Classes

- At the top of the hierarchy are two abstract classes: **`InputStream`** and **`OutputStream`**
  - All input streams are subclasses of **`InputStream`**
  - All output streams are subclasses of **`OutputStream`**
- Because they are byte-oriented they are suitable for reading binary and ASCII data (single-byte characters)
  - Use character oriented streams for multi-byte Unicode characters

Character Stream Classes

- At the top of the hierarchy are two abstract classes: **`Reader`** and **`Writer`**
  - All writers are subclasses of **`Writer`**
  - All readers are subclasses of **`Reader`**
- Readers and Writers support a wide variety of character encodings including multi-byte encodings like Unicode
  - Useful when reading and writing text (character data)
  - Supporting international character sets

Reading and Writing Files

- Lowe level byte and character streams can be used for reading and writing files
  - We start by learning about this basic form of file IO
  - It is however inflexible for reading and writing other types of data such as **`int`s**, **`double`s**, or **`String`s**
- Java provides high level stream classes for reading and writing files
  - These classes “warp” the low level stream classes to provide extended functionality to handle various types of data
  - We will learn how to use these classes for reading and writing formatted text and binary data

Reading Files Using Byte Streams

- The class **`FileInputStream`** is a stream class that can be used to read from a file
  - An object of the class **`FileInputStream`** has the method **`read()`** to read byte(s) from a file

- All the file I/O classes that follow are in the package **`java.io`**, so a program using **`FileInpuStream`** will start with the **`import`** statement:

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

Reading Files Using Byte Streams

- The process of connecting a stream to a file is called *opening the file*

- A file is open for input by creating a **`FileInputStream`** object. A commonly used constructor is

```
FileInputStream(String fileName)
           throws FileNotFoundException
```

  - ***`filename`*** specifies the name of the file to open

- A stream of the class **`FileInputStream`** is created and connected to a file as follows:

```
FileInputStream fin = new FileInputStream(fileName);
```

  - This opens the file for reading

Reading Files Using Byte Streams

- After these statements, the method **`read()`** can be used to read from the file
- The simplest form of the method **`read()`**is:

```
int read() throws IOException
```

  - reads a single byte from the file and returns it as an integer value
  - returns **`-1`** if the stream has ended
  - throws **`IOException`** if an I/O error occurs

Reading Files Using Byte Streams

- When a program is finished with a file, it should always close the stream connected to that file

```
    fin.close();
```

  - This allows the system to release any resources used to connect the stream to the file
  - If the program does not close the file before the program ends, Java will close it automatically, but it is safer to close it explicitly

Reading Files Using Byte Streams

- A program using a **`FileInputStream`** object in this way may throw two kinds of exceptions
  - An attempt to open the file may throw a **`FileNotFoundException`**
  - An invocation of **`read()`** may throw an **`IOException`**
  - Both of these exceptions should be handled
- The class **`IOException`** is the root class for a variety of exception classes having to do with input and/or output
  - These exception classes are all checked exceptions
  - Therefore, they must be caught or declared in a throws clause

Reading Files Using Byte Streams

- Note file reading program
  `class ShowFile`
- Reads from file, displays on screen
- Note
  - Statement which opens the file
  - Use of try-catch to open the file
  - Use of finally to close the file
  - Checking for the end of file

File Names

- The rules for how file names should be formed depend on a given operating system, not Java
  - When a file name is given to a java constructor for a stream, it is just a string, not a Java identifier (e.g., **`"fileName.txt"`**)
  - Any suffix used, such as **`.txt`** has no special meaning to a Java program

Path Names

- When a file name is used as an argument to a constructor for opening a file, it is assumed that the file is in the same directory or folder as the one in which the program is run
- If it is not in the same directory, the full or relative path name must be given

Path Names

- A *path name* not only gives the name of the file, but also the directory or folder in which the file exists
- A *full path name* gives a complete path name, starting from the root directory
- A *relative path name* gives the path to the file, starting with the directory in which the Java program is located

Path Names

- The way path names are specified depends on the operating system
  - A typical UNIX path name that could be used as a file name argument is

    ```
    "/user/sallyz/data/data.txt"
    ```

  - A `FileInputStream` object connected to this file is created as follows:

    ```
    FileInputStream inputStream = new
      FileInputStream("/user/sallyz/data/data.txt");
    ```

Path Names

- The Windows operating system specifies path names in a different way
  - A typical Windows path name is the following:

    ```
    C:\dataFiles\goodData\data.txt
    ```

  - A `FileInputStream` object connected to this file is created as follows:

    ```
    FileInputStream inputStream = new
      FileInputStream("C:\\dataFiles\\goodData\\data.txt");
    ```

  - Note that in Windows, `\\` must be used in place of `\`, since a single backslash denotes an the beginning of an escape sequence

Path Names

- Problems with escape characters can be avoided altogether by always using UNIX conventions when writing a path name
  - A Java program will accept a path name written in either Windows or Unix format regardless of the operating system on which it is run

Writing Files Using Byte Streams

- The class **`FileOutputStream`** is a stream class that can be used to write to a file
  - An object of the class **`FileOutputStream`** has the method **`write()`** to write to a file

- The simplest form of the method **`write()`** is

```
void write(int byteVal) throws IOException
```

  - Writes the byte specified by **`byteVal`** to the file
  - Only the low-order 8 bits are written to the file

Writing Files Using Byte Streams

- A file is open for output by creating a **`FileOutputStream`** object. Two commonly used constructors are

  `FileOutputStream(String fileName)`
  When the file is opened, any existing file with the same name is destroyed. If there is no existing file then a new file is created.

  `FileOutputStream(String filename, Boolean append)`
  If append is true, the output is appended to the end of the file. If the file does not exist then a new file is created.

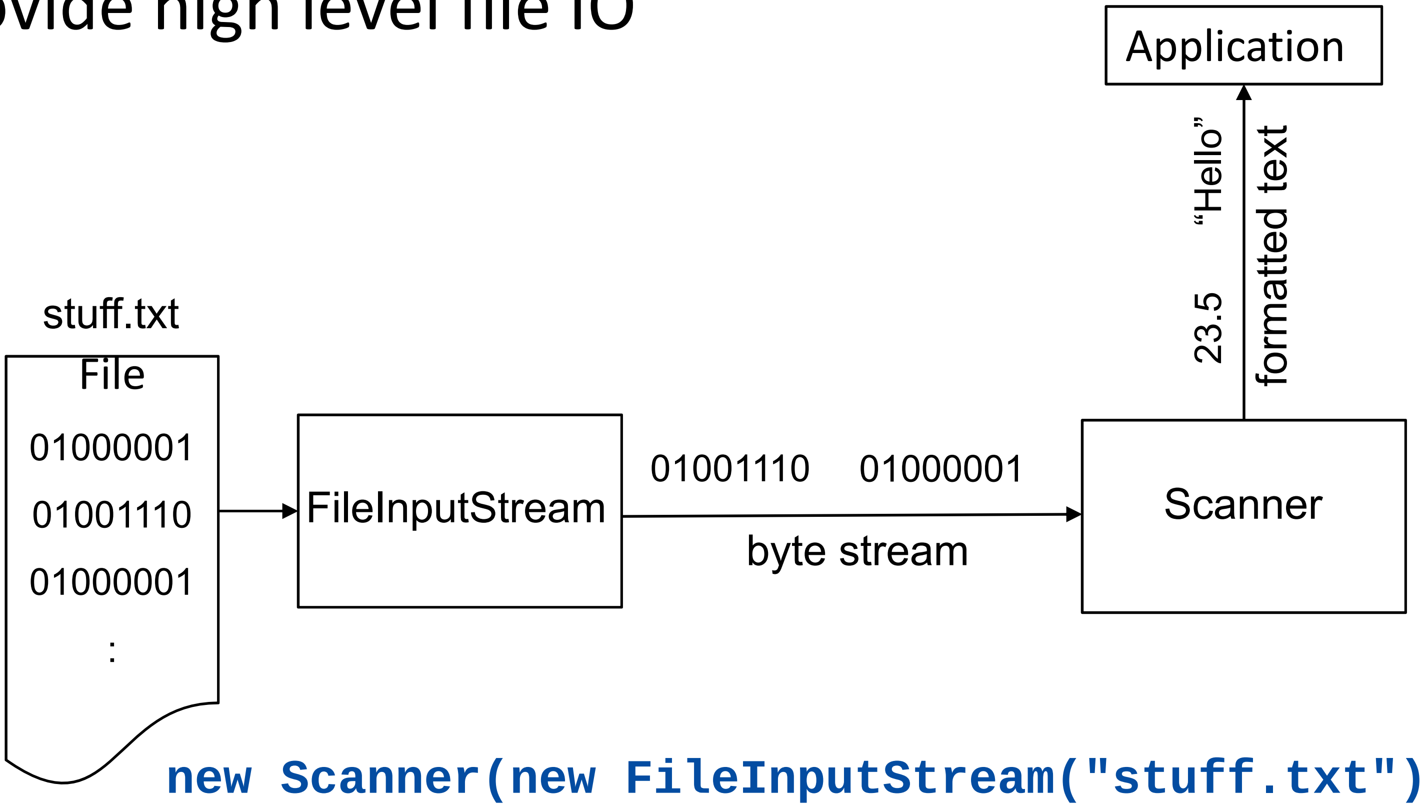
  – ***`filename`*** specifies the name of the file to open
  – ***`FileNotFoundException`*** is thrown if the file cannot be created

Writing Files Using Byte Streams

- Note file writing program
  `class CopyFile`
- Reads from file, copies to another file
- Note
  - Use of try-catch to open files
  - Use of finally to close files
  - Checking for the end of file

Reading and Writing Files: Beyond Bytes

- Java has several classes that use byte streams to provide high level file IO



```
new Scanner(new FileInputStream("stuff.txt"))
```

Nested Constructor Invocations

- Each of the Java I/O library classes serves only one function, or a small number of functions
  - Normally two or more class constructors are combined to obtain full functionality
- Therefore, expressions with two constructors are common when dealing with Java I/O classes

Nested Constructor Invocations

```
new Scanner(new FileInputStream("stuff.txt"))
```

- Above, the anonymous `FileInputStream` object establishes a connection with the `stuff.txt` file
  - However, it provides only very primitive methods for input
- The constructor for `Scanner` takes this `FileInputStream` object and adds a richer collection of input methods
  - This transforms the inner object into an instance variable of the outer object

Writing to a Text File

- The class **`PrintWriter`** is a stream class that can be used to write to a text file
  - An object of the class **`PrintWriter`** has the methods **`print`** and **`println`**
  - These are similar to the **`System.out`** methods of the same names, but are used for text file output, not screen output

Writing to a Text File

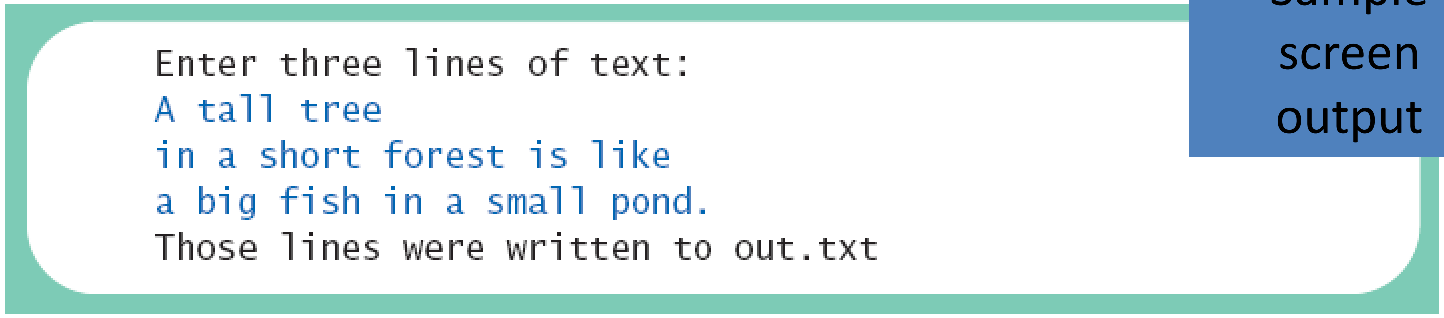
- A stream of the class **`PrintWriter`** is created and connected to a text file for writing as follows:

```
PrintWriter outputStreamName;
outputStreamName = new PrintWriter(new
                   FileOutputStream(FileName));
```

  - The class **`FileOutputStream`** takes a string representing the file name as its argument
  - The class **`PrintWriter`** takes the anonymous **`FileOutputStream`** object as its argument

Writing to a Text File

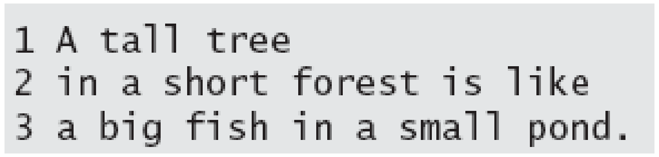
- View sample program
  class TextFileOutputDemo



```
Enter three lines of text:
A tall tree
in a short forest is like
a big fish in a small pond.
Those lines were written to out.txt
```

Sample screen output

Resulting File



```
1 A tall tree
2 in a short forest is like
3 a big fish in a small pond.
```

You can use a text editor to read this file.

Writing to a Text File

- Output streams connected to files are usually *buffered*
  - Rather than physically writing to the file as soon as possible, the data is saved in a temporary location (*buffer*)
  - When enough data accumulates, or when the method `flush` is invoked, the buffered data is written to the file all at once
  - This is more efficient, since physical writes to a file can be slow

Appending to a Text File

- To create a **`PrintWriter`** object and connect it to a text file for *appending*, a second argument, set to **`true`**, must be used in the constructor for the **`FileOutputStream`** object

```
outputStreamName = new PrintWriter(new
  FileOutputStream(FileName, true));
```

  - After this statement, the methods **`print`**, **`println`** and/or **`printf`** can be used to write to the file
  - The new text will be written *after the old text* in the file

`toString` Helps with Text File Output

- If a class has a suitable **`toString()`** method, and **`anObject`** is an object of that class, then **`anObject`** can be used as an argument to **`System.out.println`**, and it will produce sensible output
- The same thing applies to the methods **`print`** and **`println`** of the class **`PrintWriter`**

```
outputStreamName.println(anObject);
```

Reading From a Text File

- The class **`Scanner`** can be used for reading from the keyboard as well as reading from a text file
  - Simply replace the argument **`System.in`** (to the **`Scanner`** constructor) with a suitable stream that is connected to the text file

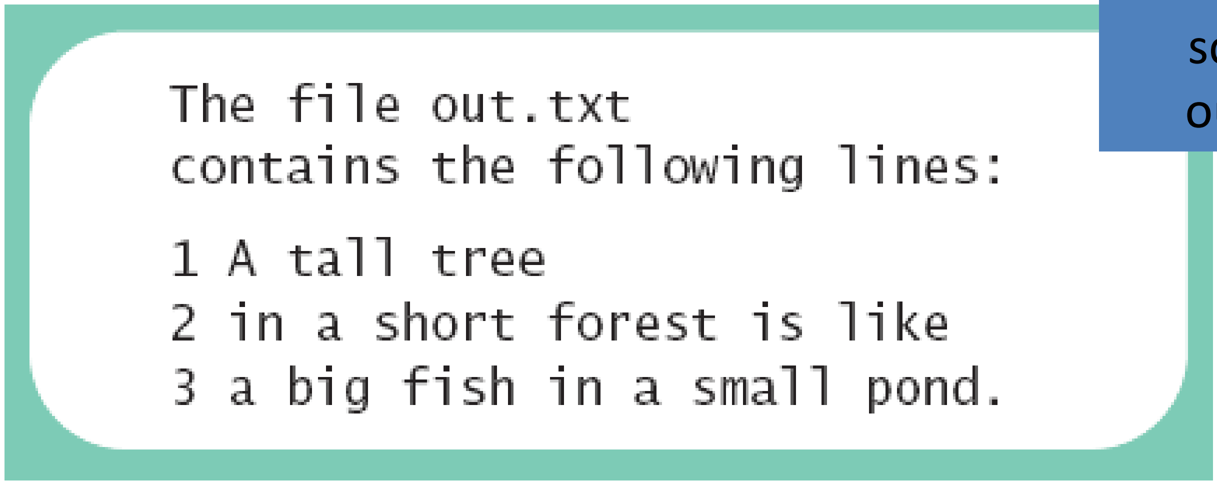
```
Scanner StreamObject =
  new Scanner(new FileInputStream(FileName));
```

- Methods of the **`Scanner`** class for reading input behave the same whether reading from the keyboard or reading from a text file
  - For example, the **`nextInt`** and **`nextLine`** methods

Reading from a Text File

- Note text file reading program
  `class TextFileInputDemo`
- Reads text from file, displays on screen
- Note
  - Statement which opens the file
  - Use of **`Scanner`** object
  - Boolean statement which reads the file and terminates reading loop

Reading from a Text File



```
The file out.txt
contains the following lines:

1 A tall tree
2 in a short forest is like
3 a big fish in a small pond.
```

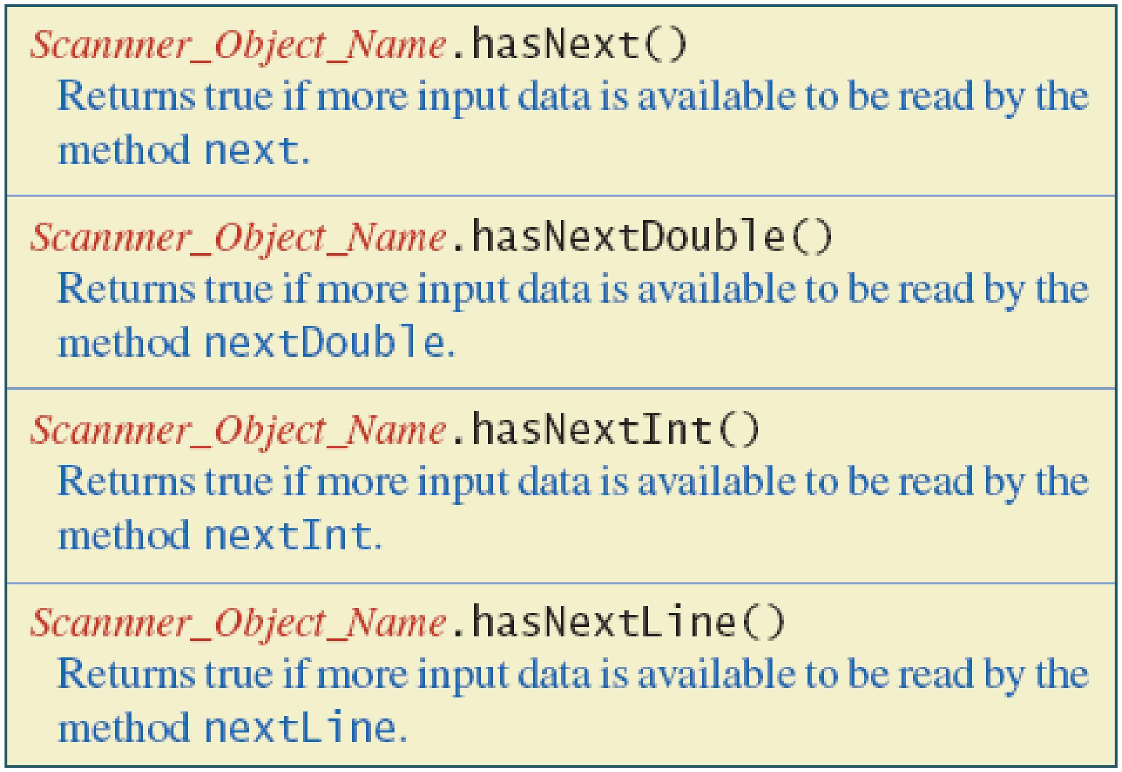
Sample screen output

Testing for the End of a Text File with **`Scanner`**

- A program that tries to read beyond the end of a file using methods of the **`Scanner`** class will cause the exception **`EOFException`** to be thrown
- However, instead of having to rely on an exception to signal the end of a file, the **`Scanner`** class provides methods such as **`hasNextInt`** and **`hasNextLine`**
  - These methods can also be used to check that the next token to be input is a suitable element of the appropriate type

Reading from a Text File

- Additional methods in class **Scanner**



Scannner_Object_Name`.hasNext()`
Returns true if more input data is available to be read by the method `next`.

Scannner_Object_Name`.hasNextDouble()`
Returns true if more input data is available to be read by the method `nextDouble`.

Scannner_Object_Name`.hasNextInt()`
Returns true if more input data is available to be read by the method `nextInt`.

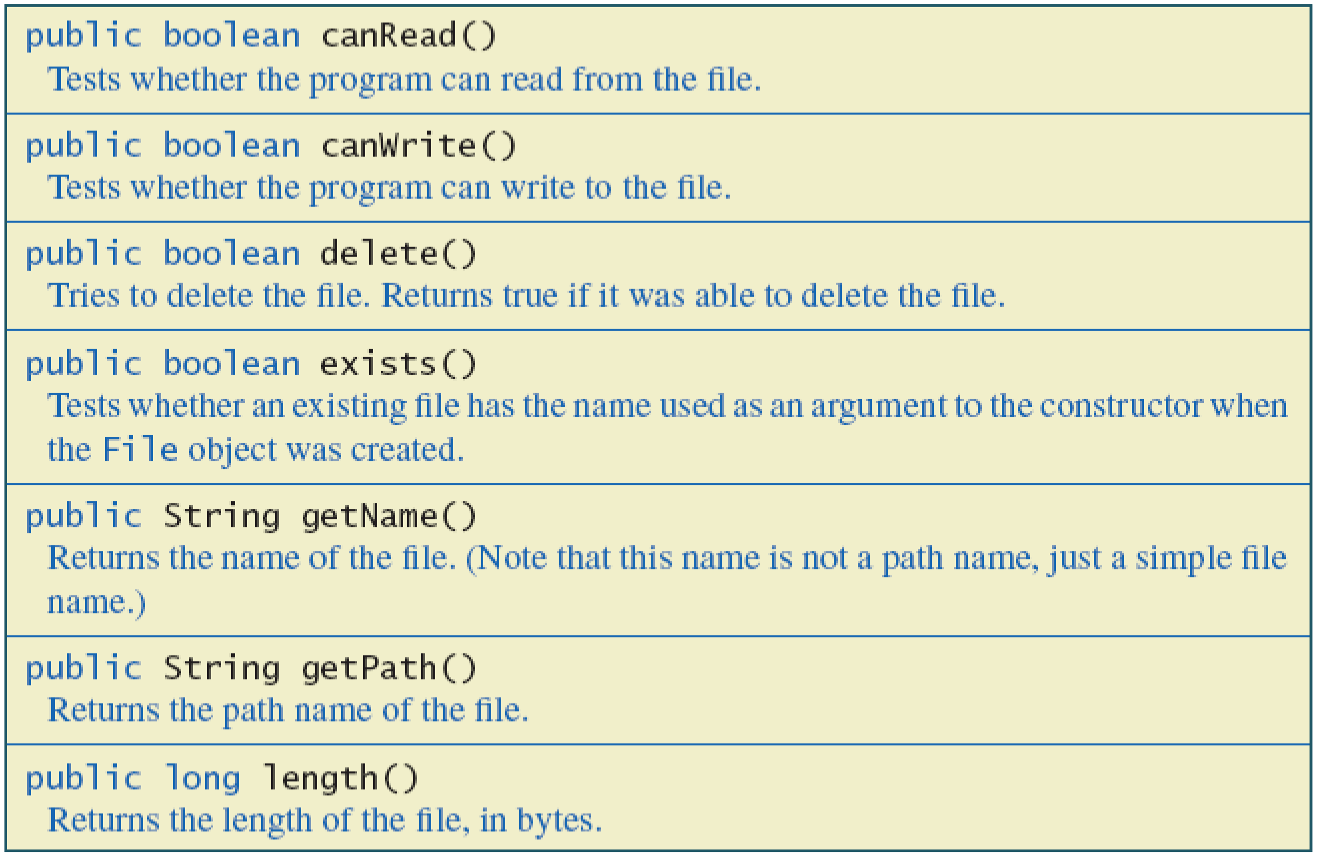
Scannner_Object_Name`.hasNextLine()`
Returns true if more input data is available to be read by the method `nextLine`.

The **`File`** Class*

- The **`File`** class is like a wrapper class for file names
  - The constructor for the class **`File`** takes a name (known as the *abstract name)* as a string argument, and produces an object that represents the file with that name
  - The **`File`** object and methods of the class **`File`** can be used to determine information about the file and its properties

Methods of the Class File

- Some methods in class **`File`**



`public boolean canRead()`	Tests whether the program can read from the file.
`public boolean canWrite()`	Tests whether the program can write to the file.
`public boolean delete()`	Tries to delete the file. Returns true if it was able to delete the file.
`public boolean exists()`	Tests whether an existing file has the name used as an argument to the constructor when the `File` object was created.
`public String getName()`	Returns the name of the file. (Note that this name is not a path name, just a simple file name.)
`public String getPath()`	Returns the path name of the file.
`public long length()`	Returns the length of the file, in bytes.

Binary Files

- Binary files store data in the same format used by computer memory to store the values of variables
  - No conversion needs to be performed when a value is stored or retrieved from a binary file
- Java binary files, unlike other binary language files, are portable
  - A binary file created by a Java program can be moved from one computer to another
  - These files can then be read by a Java program, but only by a Java program

Writing Simple Data to a Binary File

- The class `ObjectOutputStream` is a stream class that can be used to write to a binary file
  - An object of this class has methods to write strings, values of primitive types, and objects to a binary file
- A program using `ObjectOutputStream` needs to import several classes from package `java.io`:

```
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
```

Opening a Binary File for Output

- An **`ObjectOutputStream`** object is created and connected to a binary file as follows:

```
ObjectOutputStream outputStreamName = new
                    ObjectOutputStream(new
                    FileOutputStream(FileName));
```

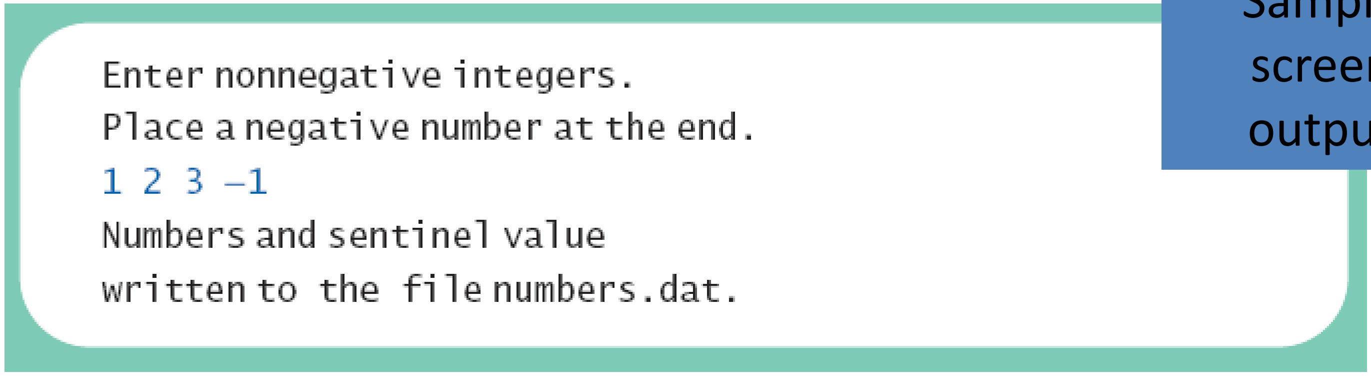
  - The constructor for **`FileOutputStream`** may throw a **`FileNotFoundException`**
  - The constructor for **`ObjectOutputStream`** may throw an **`IOException`**
  - Each of these must be handled

Opening a Binary File for Output

- After opening the file, **`ObjectOutputStream`** methods can be used to write to the file
  - Methods used to output primitive values include **`writeInt`**, **`writeDouble`**, **`writeChar`**, and **`writeBoolean`**
- *UTF* is an encoding scheme used to encode Unicode characters that favors the ASCII character set
  - The method **`writeUTF`** can be used to output values of type **`String`**
- The stream should always be closed after writing

Creating a Binary File

- View program which writes integers, **`class BinaryOutputDemo`**

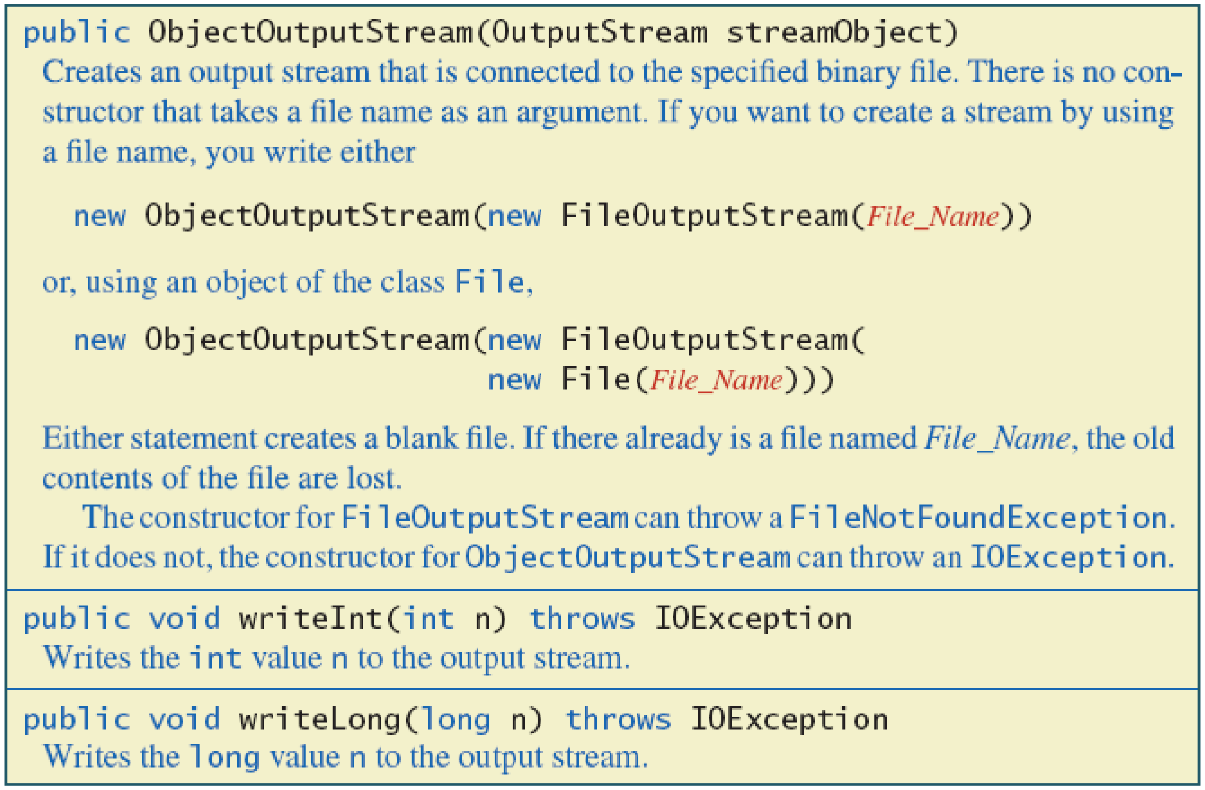


```
Enter nonnegative integers.
Place a negative number at the end.
1 2 3 -1
Numbers and sentinel value
written to the file numbers.dat.
```

Sample screen output

Writing Primitive Values to a Binary File

- Some methods in class **`ObjectOutputStream`**



```
public ObjectOutputStream(OutputStream streamObject)
```

Creates an output stream that is connected to the specified binary file. There is no constructor that takes a file name as an argument. If you want to create a stream by using a file name, you write either

```
new ObjectOutputStream(new FileOutputStream(File_Name))
```

or, using an object of the class `File`,

```
new ObjectOutputStream(new FileOutputStream(
                       new File(File_Name)))
```

Either statement creates a blank file. If there already is a file named *File_Name*, the old contents of the file are lost.

The constructor for `FileOutputStream` can throw a `FileNotFoundException`. If it does not, the constructor for `ObjectOutputStream` can throw an `IOException`.

```
public void writeInt(int n) throws IOException
```

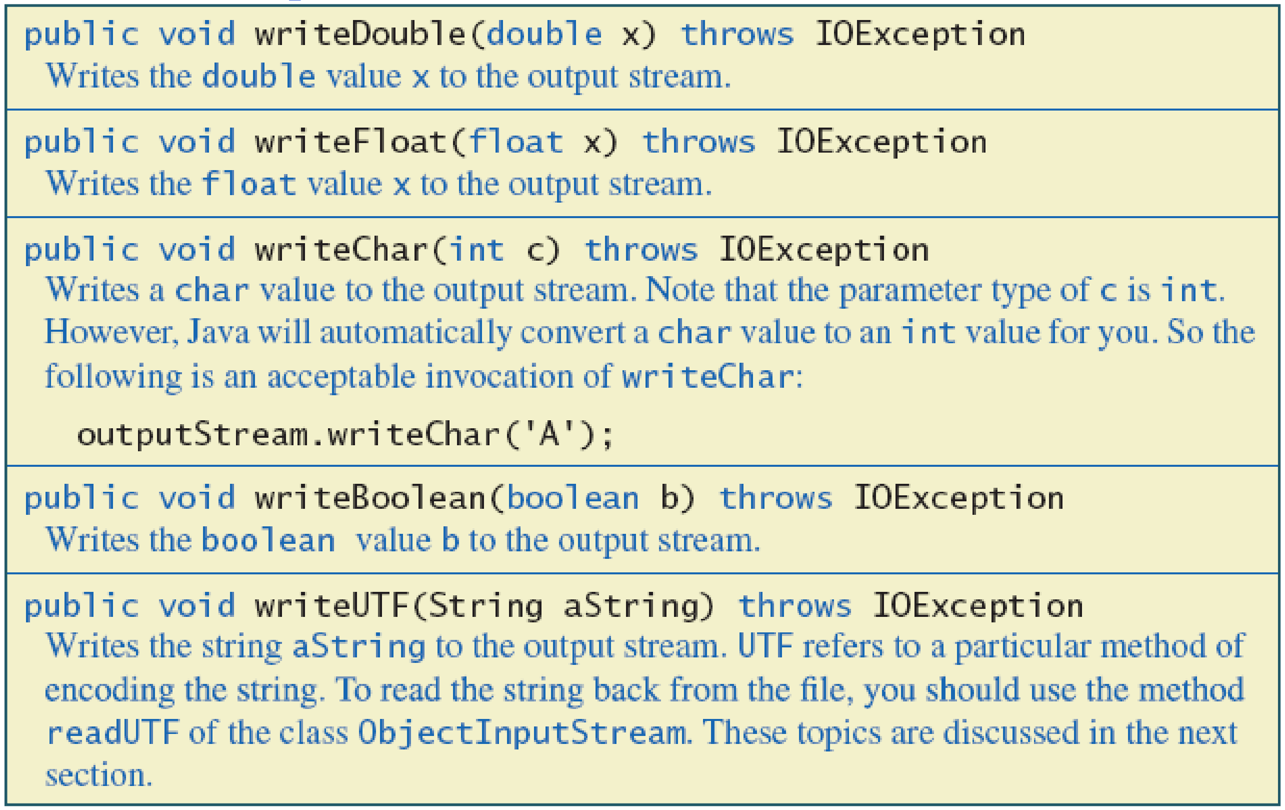
Writes the `int` value `n` to the output stream.

```
public void writeLong(long n) throws IOException
```

Writes the `long` value `n` to the output stream.

Writing Primitive Values to a Binary File

- Some methods in class **`ObjectOutputStream`**



`public void writeDouble(double x) throws IOException`
Writes the `double` value `x` to the output stream.

`public void writeFloat(float x) throws IOException`
Writes the `float` value `x` to the output stream.

`public void writeChar(int c) throws IOException`
Writes a `char` value to the output stream. Note that the parameter type of `c` is `int`. However, Java will automatically convert a `char` value to an `int` value for you. So the following is an acceptable invocation of `writeChar`:

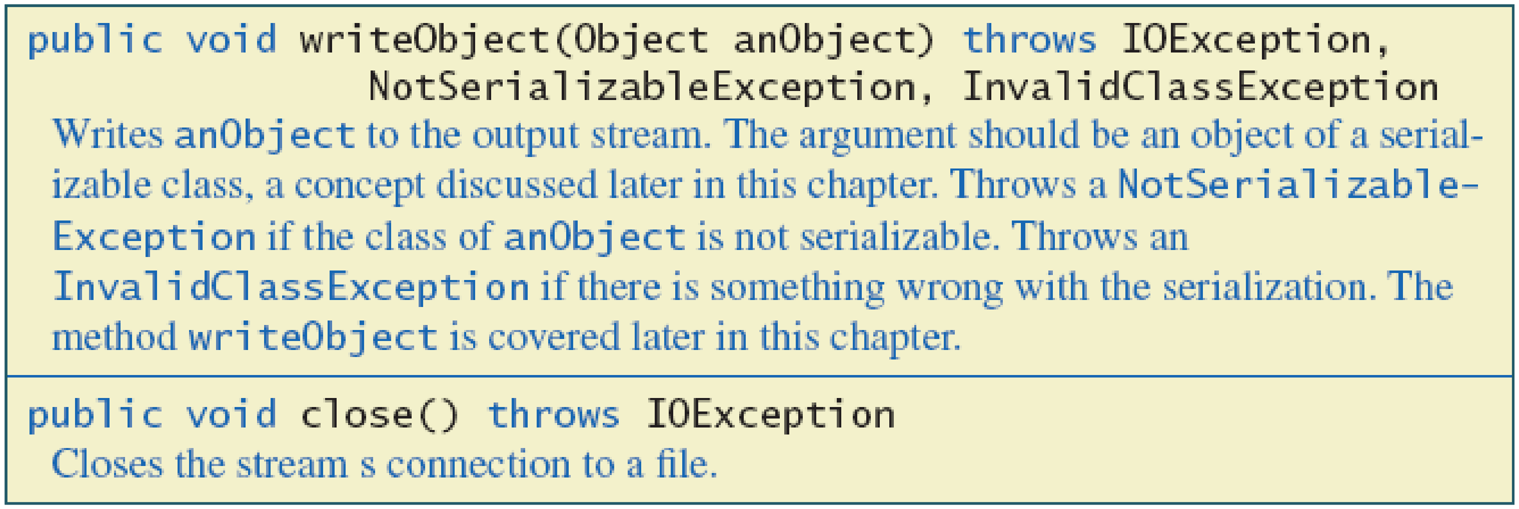
```
outputStream.writeChar('A');
```

`public void writeBoolean(boolean b) throws IOException`
Writes the `boolean` value `b` to the output stream.

`public void writeUTF(String aString) throws IOException`
Writes the string `aString` to the output stream. UTF refers to a particular method of encoding the string. To read the string back from the file, you should use the method `readUTF` of the class `ObjectInputStream`. These topics are discussed in the next section.

Writing Primitive Values to a Binary File

- Some methods in class **`ObjectOutputStream`**



```
public void writeObject(Object anObject) throws IOException,
                    NotSerializableException, InvalidClassException
```

Writes `anObject` to the output stream. The argument should be an object of a serializable class, a concept discussed later in this chapter. Throws a `NotSerializableException` if the class of `anObject` is not serializable. Throws an `InvalidClassException` if there is something wrong with the serialization. The method `writeObject` is covered later in this chapter.

```
public void close() throws IOException
```

Closes the stream s connection to a file.

Reading Simple Data from a Binary File

- The class `ObjectInputStream` is a stream class that can be used to read from a binary file
  - An object of this class has methods to read strings, values of primitive types, and objects from a binary file
- A program using `ObjectInputStream` needs to import several classes from package `java.io`:

```
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
```

Opening a Binary File for Reading

- An **`ObjectInputStream`** object is created and connected to a binary file as follows:

```
ObjectInputStream inStreamName = new
                     ObjectInputStream(new
                     FileInputStream(FileName));
```

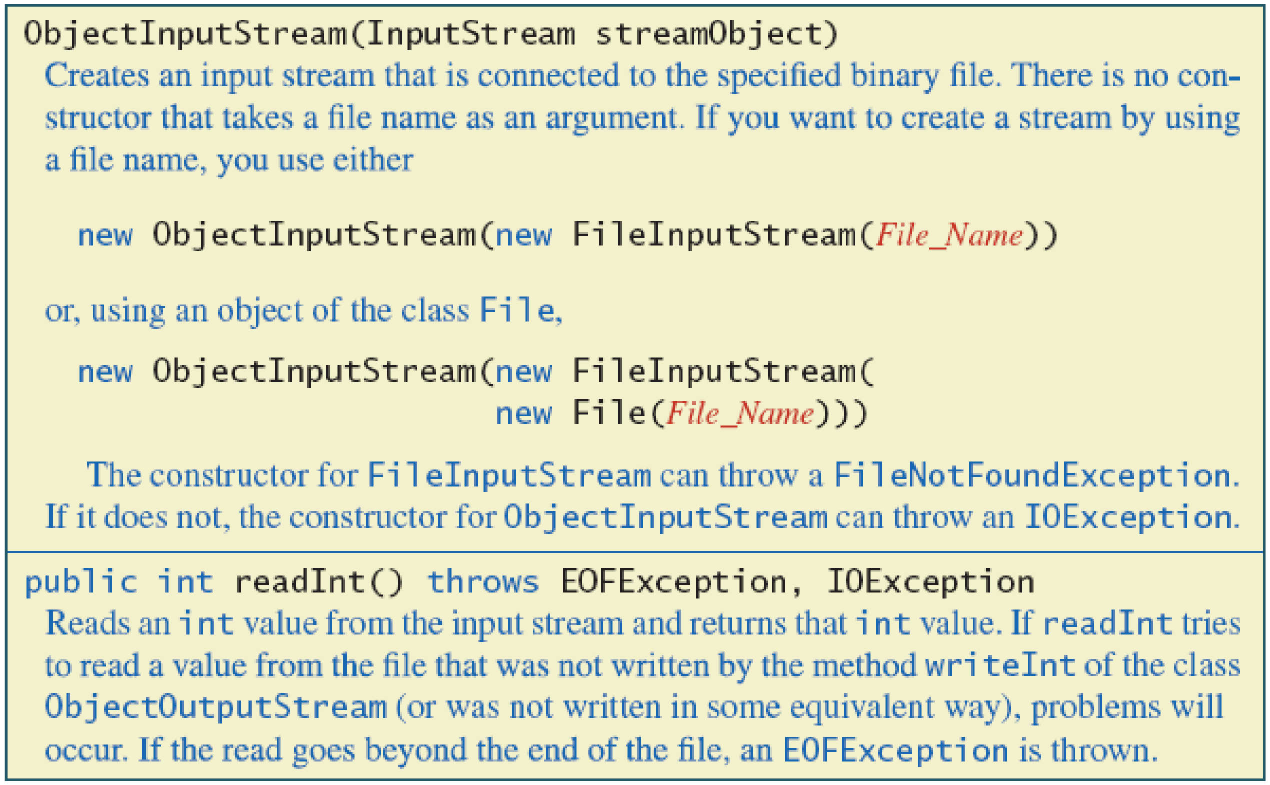
  - The constructor for **`FileInputStream`** may throw a **`FileNotFoundException`**
  - The constructor for **`ObjectInputStream`** may throw an **`IOException`**
  - Each of these must be handled

Opening a Binary File for Reading

- After opening the file, `ObjectInputStream` methods can be used to read to the file
  - Methods used to input primitive values include `readInt`, `readDouble`, `readChar`, and `readBoolean`
  - The method `readUTF` is used to input values of type `String`
- If the file contains multiple types, each item type must be read in exactly the same order it was written to the file
- The stream should be closed after reading

Reading from a Binary File

- Some methods of class **`ObjectInputStream`**



```
ObjectInputStream(InputStream streamObject)
```

Creates an input stream that is connected to the specified binary file. There is no constructor that takes a file name as an argument. If you want to create a stream by using a file name, you use either

```
new ObjectInputStream(new FileInputStream(File_Name))
```

or, using an object of the class `File`,

```
new ObjectInputStream(new FileInputStream(
                          new File(File_Name)))
```

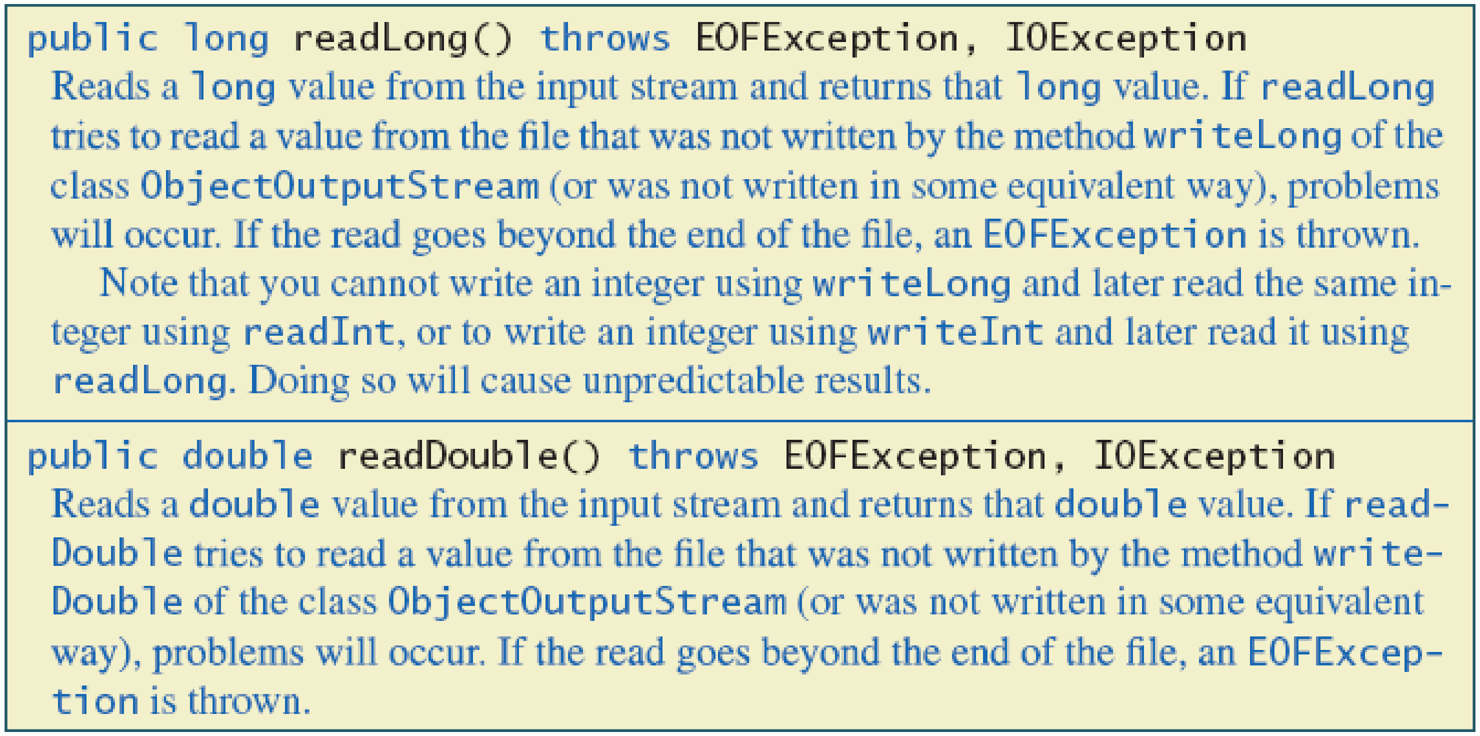
The constructor for `FileInputStream` can throw a `FileNotFoundException`. If it does not, the constructor for `ObjectInputStream` can throw an `IOException`.

```
public int readInt() throws EOFException, IOException
```

Reads an `int` value from the input stream and returns that `int` value. If `readInt` tries to read a value from the file that was not written by the method `writeInt` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

Reading from a Binary File

- Some methods of class `ObjectInputStream`



`public long readLong() throws EOFException, IOException`

Reads a `long` value from the input stream and returns that `long` value. If `readLong` tries to read a value from the file that was not written by the method `writeLong` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

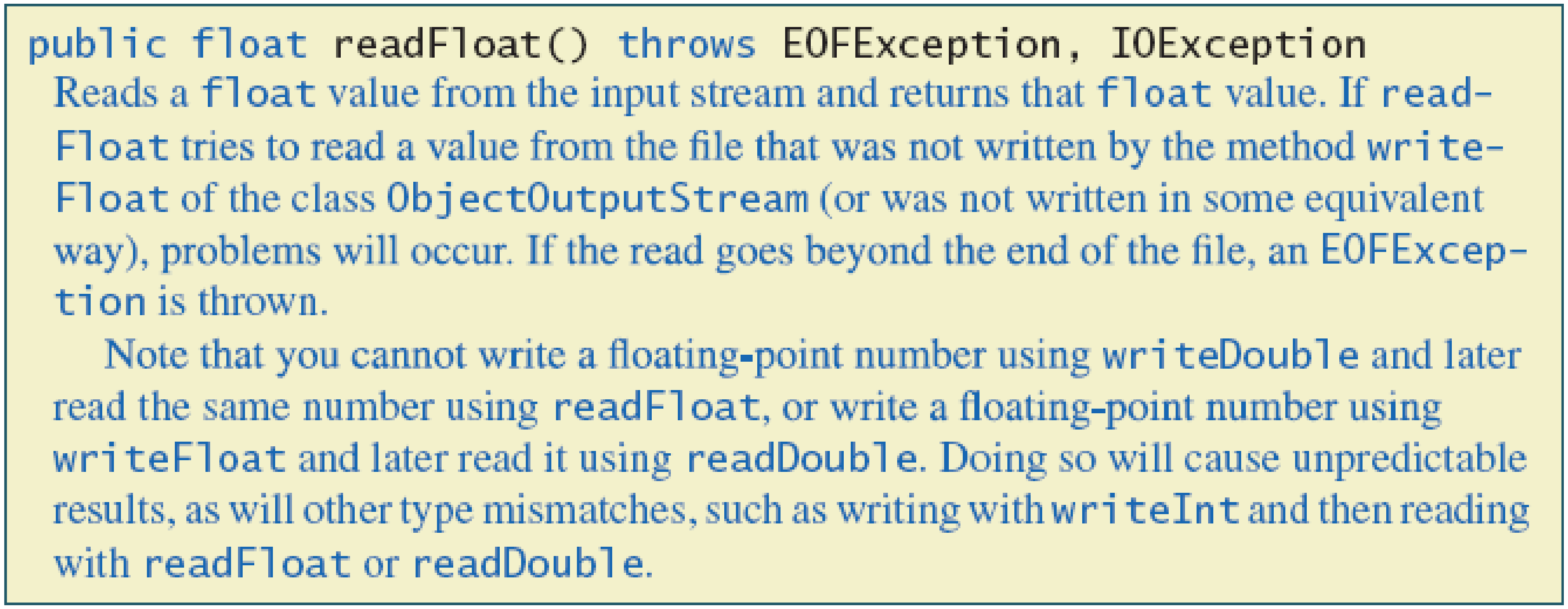
Note that you cannot write an integer using `writeLong` and later read the same integer using `readInt`, or to write an integer using `writeInt` and later read it using `readLong`. Doing so will cause unpredictable results.

`public double readDouble() throws EOFException, IOException`

Reads a `double` value from the input stream and returns that `double` value. If `readDouble` tries to read a value from the file that was not written by the method `writeDouble` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

Reading from a Binary File

- Some methods of class **`ObjectInputStream`**



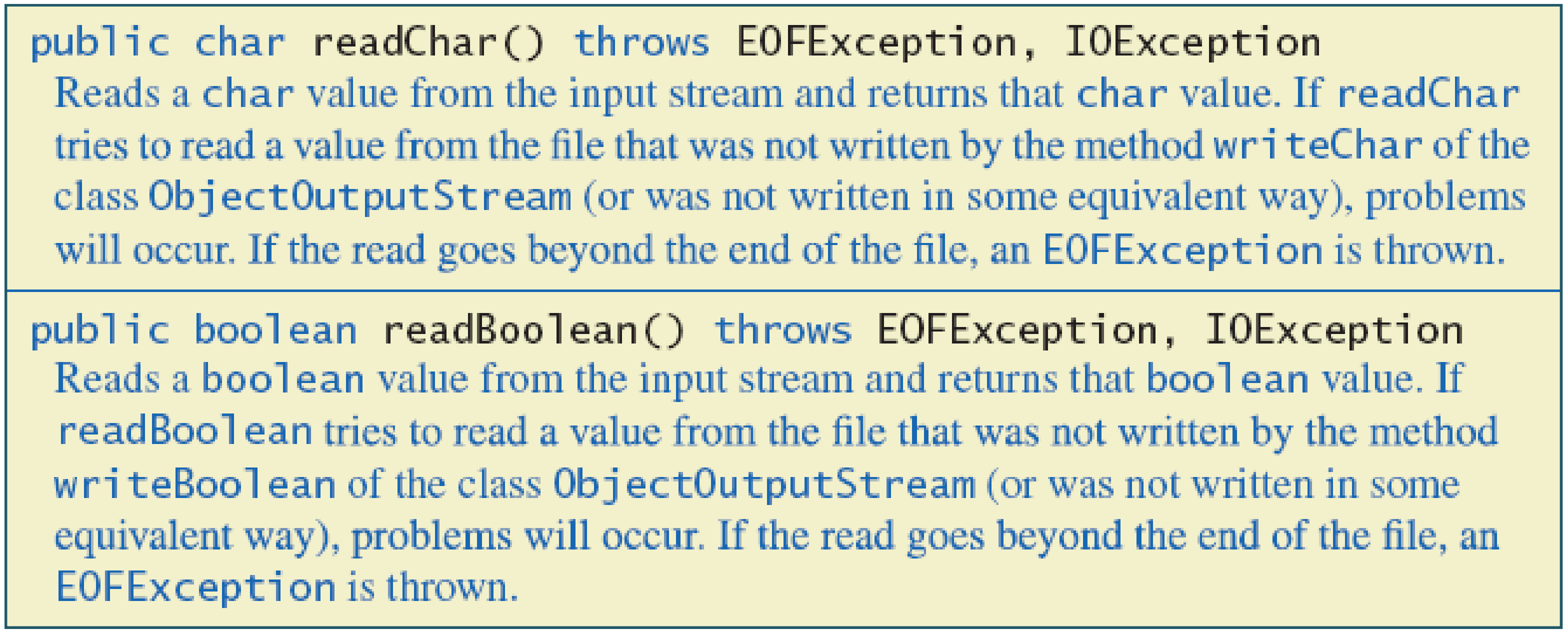
`public float readFloat() throws EOFException, IOException`

Reads a `float` value from the input stream and returns that `float` value. If `readFloat` tries to read a value from the file that was not written by the method `writeFloat` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

Note that you cannot write a floating-point number using `writeDouble` and later read the same number using `readFloat`, or write a floating-point number using `writeFloat` and later read it using `readDouble`. Doing so will cause unpredictable results, as will other type mismatches, such as writing with `writeInt` and then reading with `readFloat` or `readDouble`.

Reading from a Binary File

- Some methods of class **`ObjectInputStream`**



`public char readChar() throws EOFException, IOException`

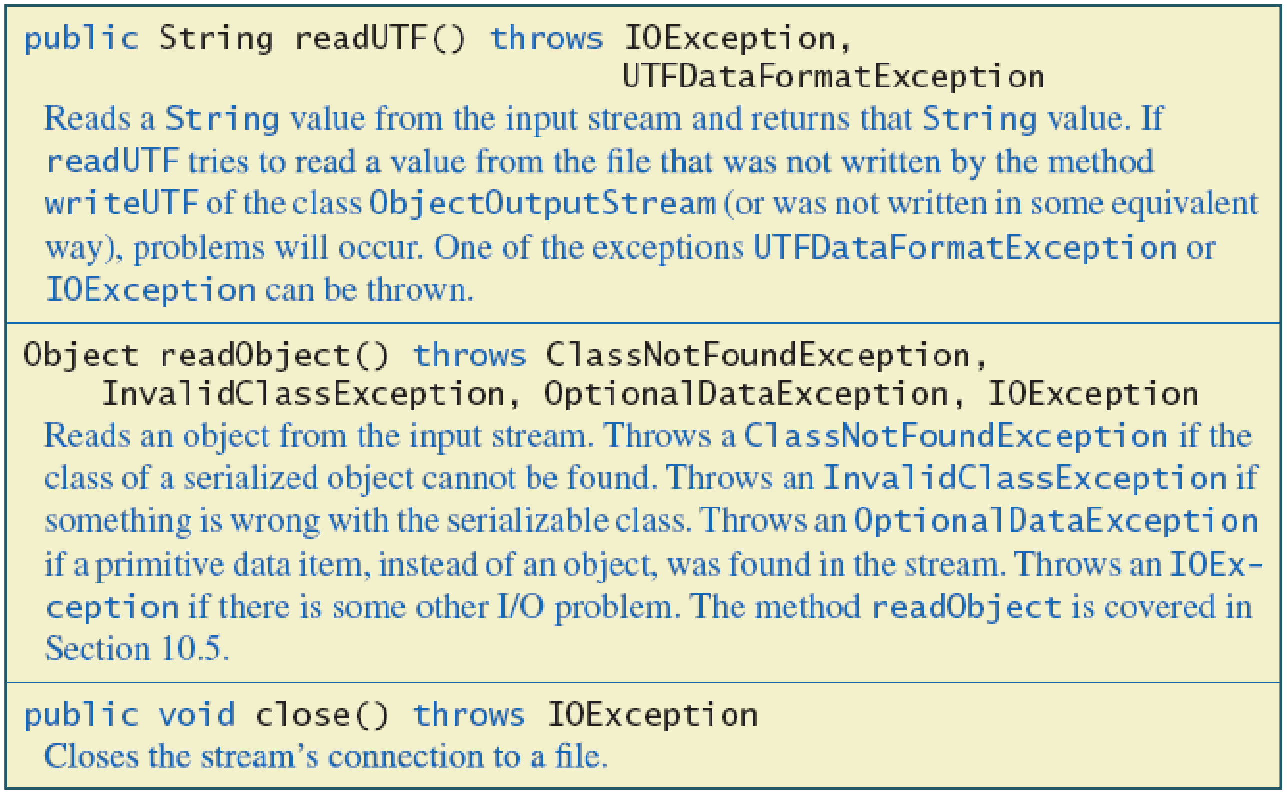
Reads a `char` value from the input stream and returns that `char` value. If `readChar` tries to read a value from the file that was not written by the method `writeChar` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

`public boolean readBoolean() throws EOFException, IOException`

Reads a `boolean` value from the input stream and returns that `boolean` value. If `readBoolean` tries to read a value from the file that was not written by the method `writeBoolean` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end of the file, an `EOFException` is thrown.

Reading from a Binary File

- Some methods of class **`ObjectInputStream`**



```
public String readUTF() throws IOException,
                               UTFDataFormatException
```

Reads a `String` value from the input stream and returns that `String` value. If `readUTF` tries to read a value from the file that was not written by the method `writeUTF` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. One of the exceptions `UTFDataFormatException` or `IOException` can be thrown.

```
Object readObject() throws ClassNotFoundException,
    InvalidClassException, OptionalDataException, IOException
```

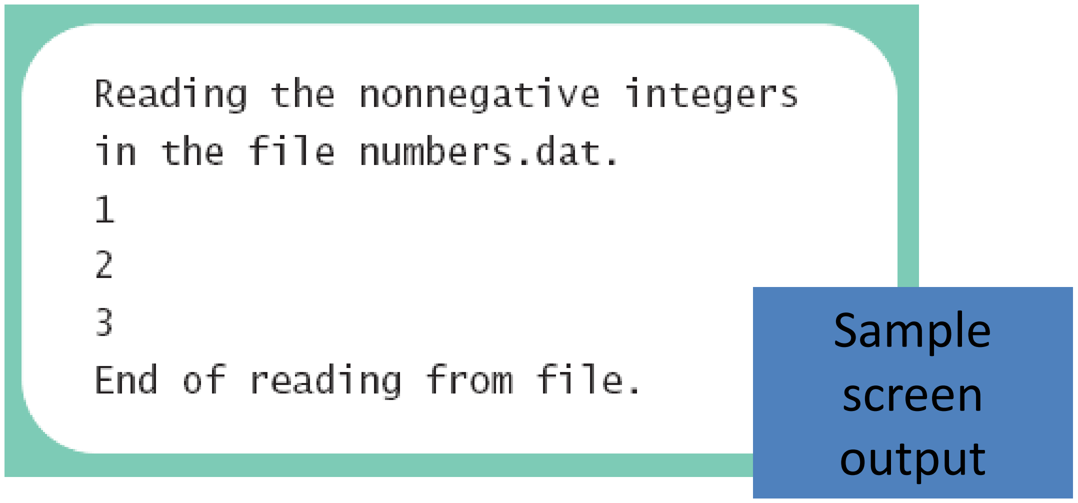
Reads an object from the input stream. Throws a `ClassNotFoundException` if the class of a serialized object cannot be found. Throws an `InvalidClassException` if something is wrong with the serializable class. Throws an `OptionalDataException` if a primitive data item, instead of an object, was found in the stream. Throws an `IOEx-ception` if there is some other I/O problem. The method `readObject` is covered in Section 10.5.

```
public void close() throws IOException
```

Closes the stream's connection to a file.

Reading from a Binary File

- View program to read
  `class BinaryInputDemo`



```
Reading the nonnegative integers
in the file numbers.dat.
1
2
3
End of reading from file.
```

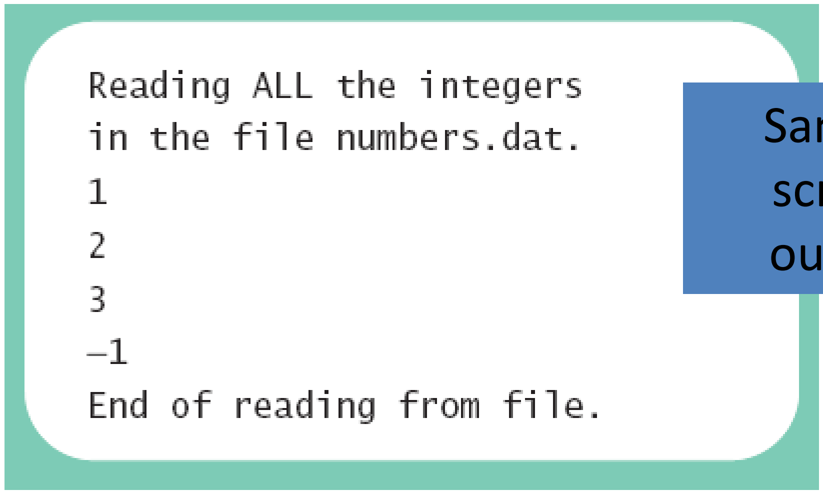
Sample screen output

Checking for the End of a Binary File the Correct Way

- All of the `ObjectInputStream` methods that read from a binary file throw an `EOFException` when trying to read beyond the end of a file
  - This can be used to end a loop that reads all the data in a file
- Note that different file-reading methods check for the end of a file in different ways
  - Testing for the end of a file in the wrong way can cause a program to go into an infinite loop or terminate abnormally

The Class **`EOFException`**

- View example program
  class **`EOFExceptionDemo`**



```
Reading ALL the integers
in the file numbers.dat.
1
2
3
-1
End of reading from file.
```

Sample screen output

Binary I/O of Objects

- Objects can also be input and output from a binary file
  - Use the **`writeObject`** method of the class **`ObjectOutputStream`** to write an object to a binary file
  - Use the **`readObject`** method of the class **`ObjectInputStream`** to read an object from a binary file
  - In order to use the value returned by **`readObject`** as an object of a class, it must be type cast first:

```
SomeClass someObject =
  (SomeClass)objectInputStream.readObject();
```

Binary I/O of Objects

- The class of the object being read or written must implement the ***`Serializable`*** *interface*
  - The **`Serializable`** interface is easy to use and requires no knowledge of interfaces
  - A class that implements the **`Serializable`** interface is said to be a *serializable class*

The **`Serializable`** Interface

- In order to make a class serializable, simply add **`implements Serializable`** to the heading of the class definition

```
public class SomeClass implements Serializable
```

- When a serializable class has instance variables of a class type, then all those classes must be serializable also
  - A class is not serializable unless the classes for all instance variables are also serializable for all levels of instance variables within classes

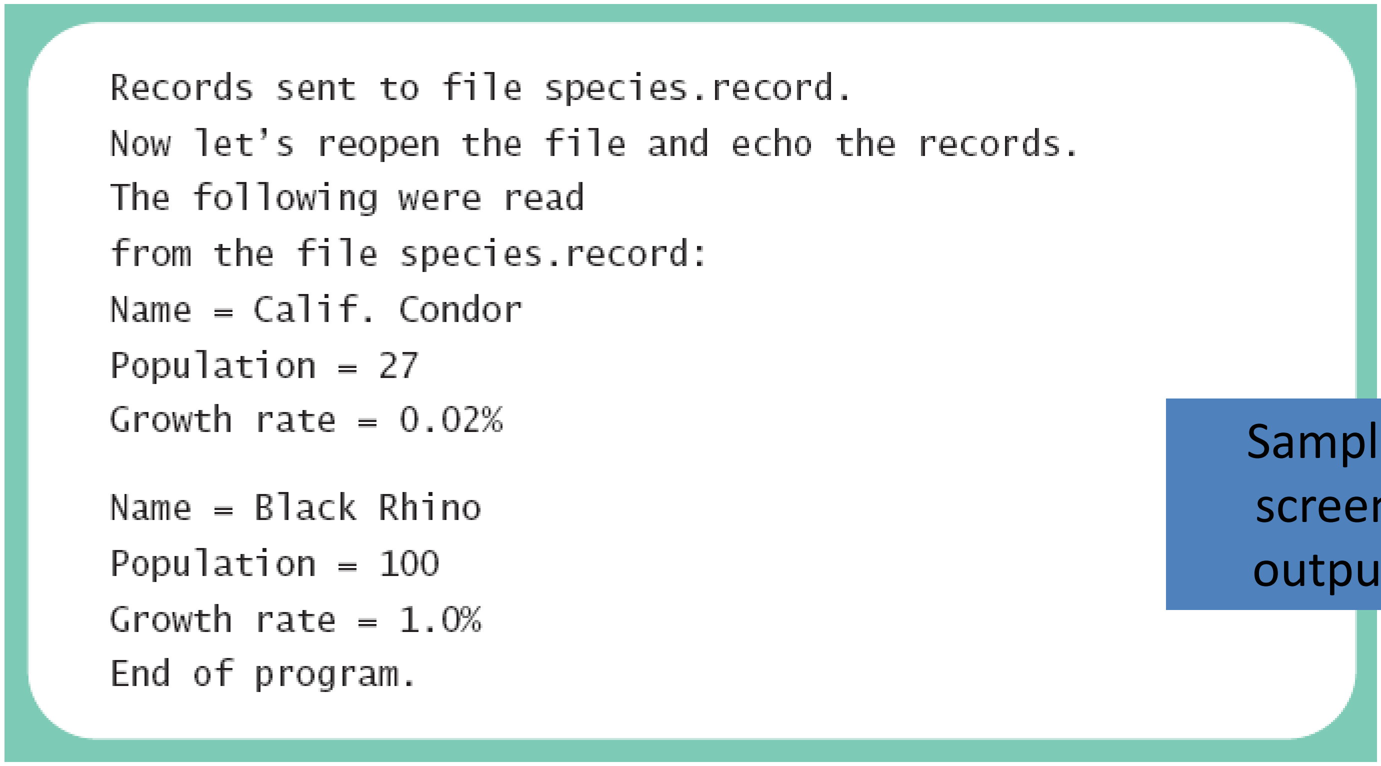
Binary-File I/O with Class Objects

- Interface **`Serializable`** is an empty interface
  - No need to implement additional methods
  - Tells Java to make the class serializable (class objects convertible to sequence of bytes)
- View sample class
  `class Species`

Binary-File I/O with Class Objects

- Once we have a class that is specified as **`Serializable`** we can write objects to a binary file
  - Use method **`writeObject`**
- Read objects with method **`readObject();`**
  - Also required to use typecast of the object
- View sample program
  `class ObjectIODemo`

Binary-File I/O with Class Objects



```
Records sent to file species.record.
Now let's reopen the file and echo the records.
The following were read
from the file species.record:
Name = Calif. Condor
Population = 27
Growth rate = 0.02%

Name = Black Rhino
Population = 100
Growth rate = 1.0%
End of program.
```

Sample screen output

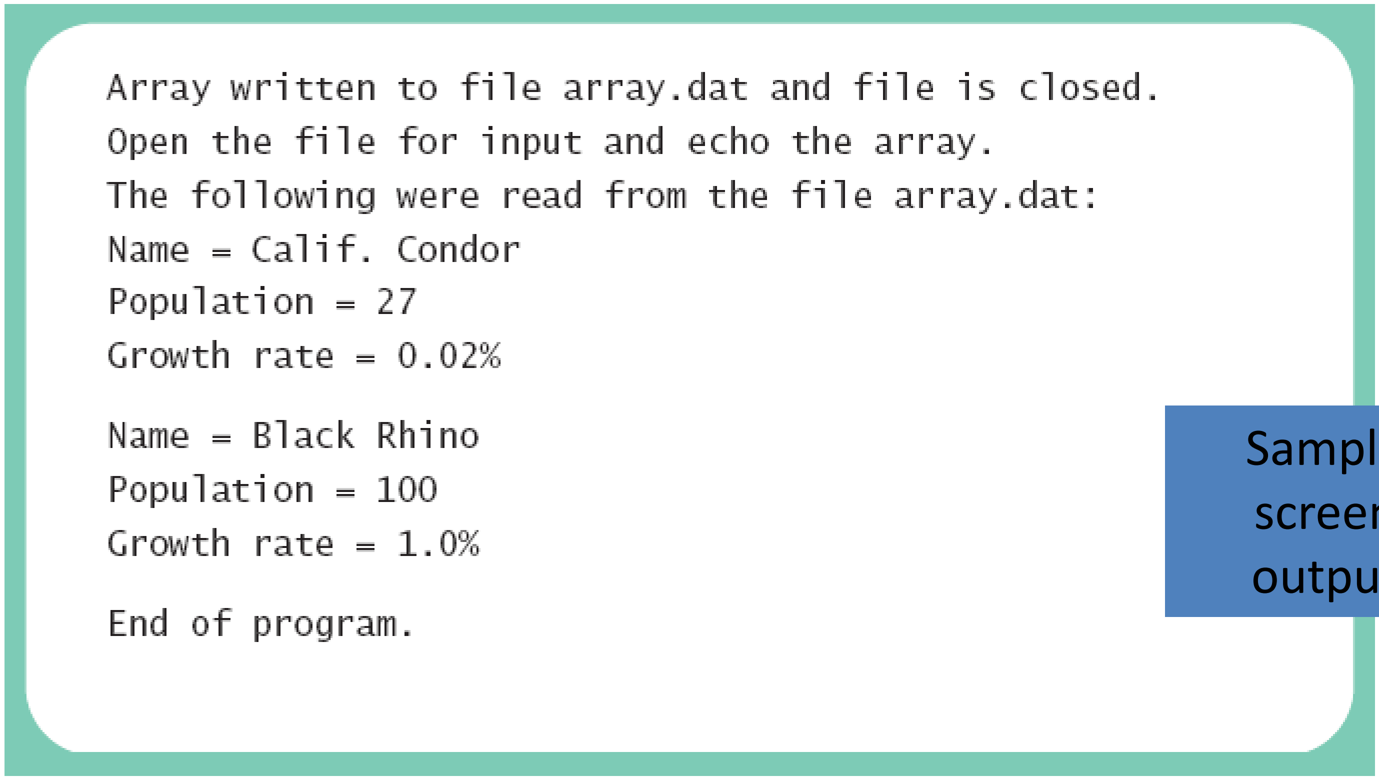
Array Objects in Binary Files

- Since an array is an object, arrays can also be read and written to binary files using **`readObject`** and **`writeObject`**
  - If the base type is a class, then it must also be serializable, just like any other class type
  - Since **`readObject`** returns its value as type **`Object`** (like any other object), it must be type cast to the correct array type:

```
SomeClass[] someObject =
 (SomeClass[])objectInputStream.readObject();
```

Array Objects in Binary Files

- View array I/O program
  `class ArrayIODemo`



```
Array written to file array.dat and file is closed.
Open the file for input and echo the array.
The following were read from the file array.dat:
Name = Calif. Condor
Population = 27
Growth rate = 0.02%

Name = Black Rhino
Population = 100
Growth rate = 1.0%

End of program.
```

Sample screen output

Summary

- Files with characters are text files
  - Other files are binary files
- Programs can use `PrintWriter` and `Scanner` for I/O
- Always check for end of file
- File name can be literal string or variable of type `String`
- Class `File` gives additional capabilities to deal with file names

Summary

- Use `ObjectOutputStream` and `ObjectInputStream` classes enable writing to, reading from binary files
- Use `writeObject` to write class objects to binary file
- Use `readObject` with type cast to read objects from binary file
- Classes for binary I/O must be serializable

- PrintStream
- BufferedReader
- DataInput/OutputStream

•Byte Streams handle I/O of raw binary data.
•Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
•Buffered Streams optimize input and output by reducing the number of calls to the native API.
•Scanning and Formatting allows a program to read and write formatted text.
•I/O from the Command Line describes the Standard Streams and the Console object.
•Data Streams handle binary I/O of primitive data type and String values.
•Object Streams handle binary I/O of objects.