## 10.4 Basic Binary File I/O

- Important classes for binary file **output** (to the file)
  - » **`ObjectOutputStream`** ; **filter class**
  - » **`FileOutputStream`**
- Important classes for binary file **input** (from the file):
  - » **`ObjectInputStream`** ; **filter class**
  - » **`FileInputStream`**
- Note that **`FileOutputStream`** and **`FileInputStream`** are used only for their constructors, which can **take file names** as arguments.
  - » **`ObjectOutputStream`** and **`ObjectInputStream`** cannot take file names as arguments for their constructors.
- To use these classes your program needs a line like the following:

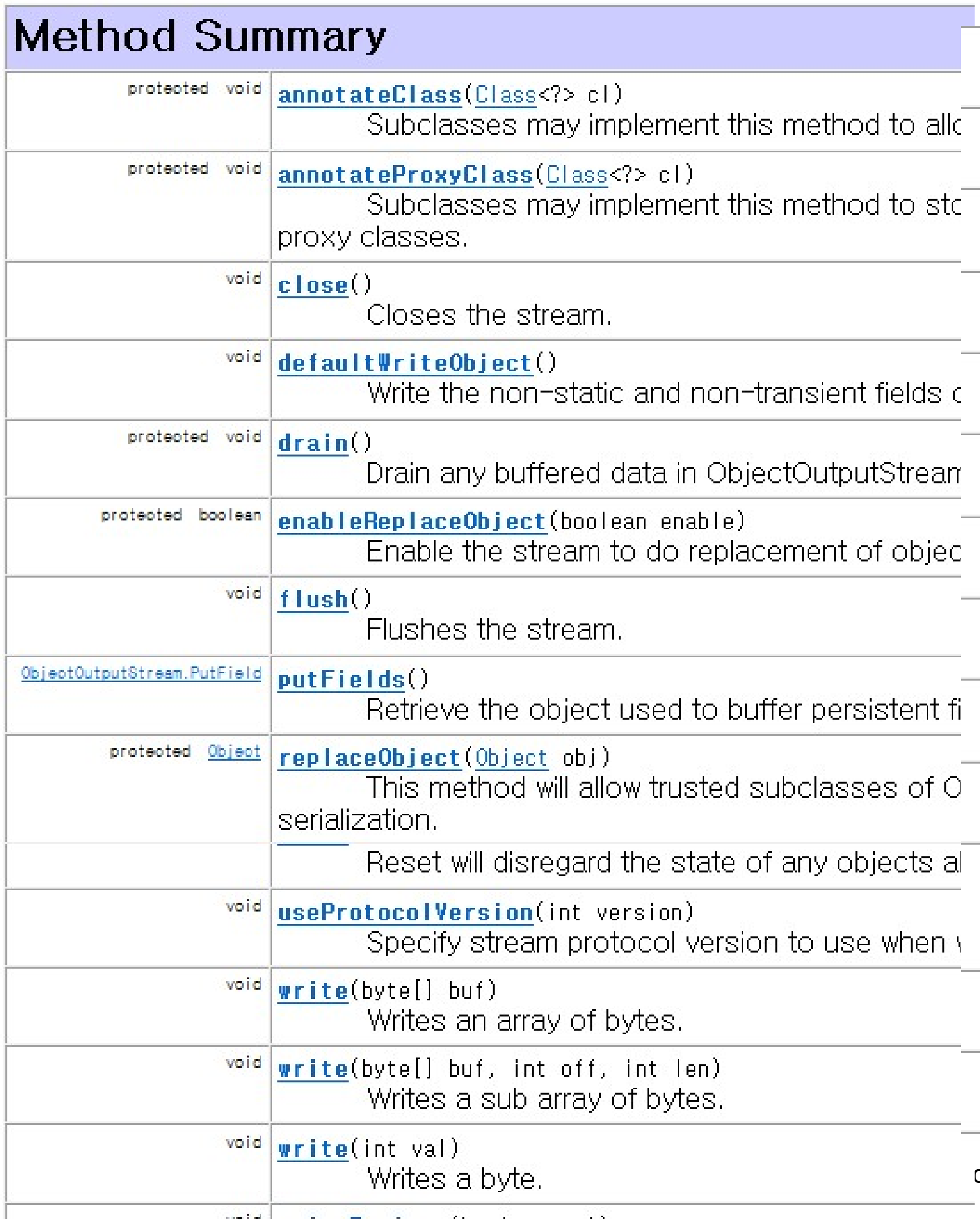
```
import java.io.*;
```

# Java File I/O: Stream Classes

- `ObjectInputStream` and `ObjectOutputStream`:
  - » have methods to either read or write data **one byte** at a time
  - » automatically convert numbers and characters into binary
    - – binary-encoded numeric files (files with numbers) are not readable by a text editor, but store data more efficiently

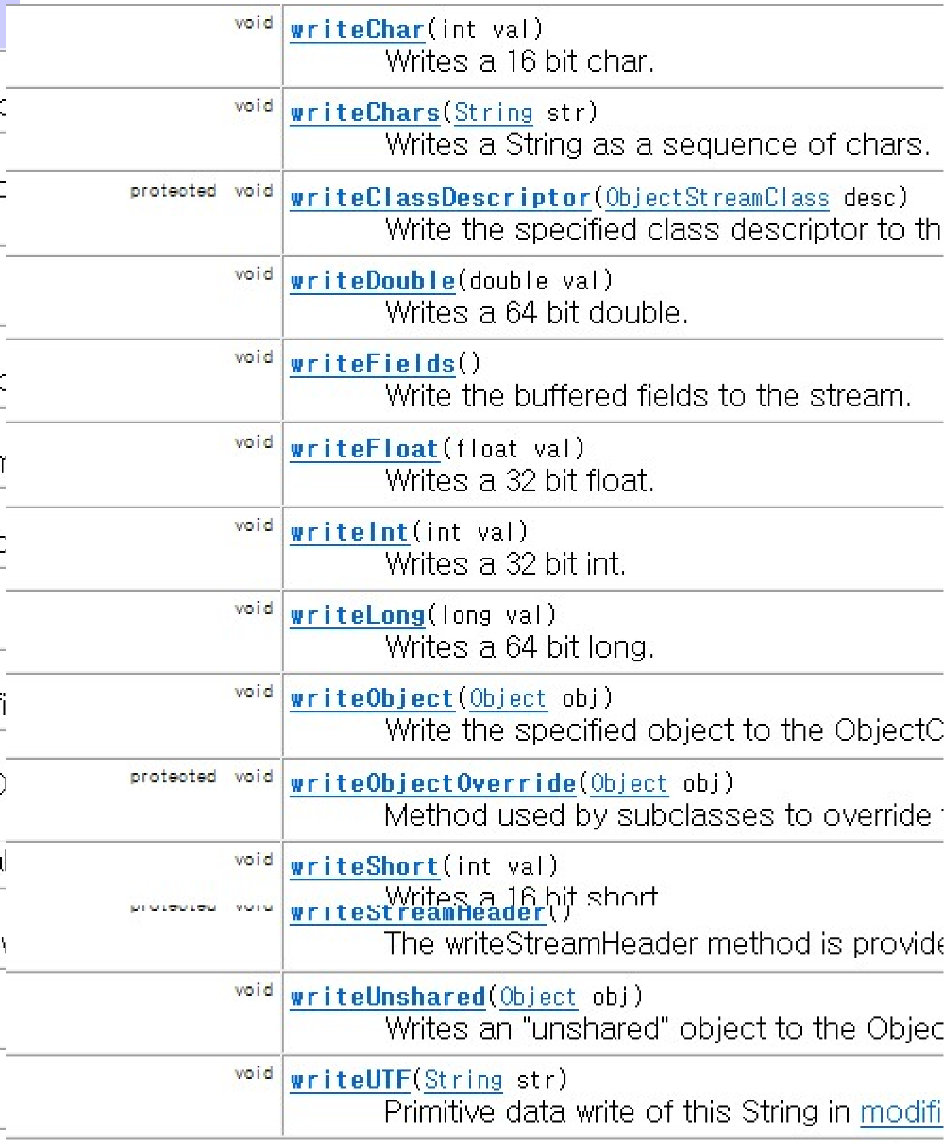
- Remember:
  - » *input* means data into a program, not the file
  - » similarly, *output* means data out of a program, not the file

# ObjectOutputStream



## Method Summary

| | |
|---|---|
| protected void | annotateClass(Class<?> cl) Subclasses may implement this method to allo |
| protected void | annotateProxyClass(Class<?> cl) Subclasses may implement this method to sto proxy classes. |
| void | close() Closes the stream. |
| void | defaultWriteObject() Write the non-static and non-transient fields c |
| protected void | drain() Drain any buffered data in ObjectOutputStream |
| protected boolean | enableReplaceObject(boolean enable) Enable the stream to do replacement of objec |
| void | flush() Flushes the stream. |
| ObjectOutputStream.PutField | putFields() Retrieve the object used to buffer persistent fi |
| protected Object | replaceObject(Object obj) This method will allow trusted subclasses of O serialization. |
| | Reset will disregard the state of any objects al |
| void | useProtocolVersion(int version) Specify stream protocol version to use when v |
| void | write(byte[] buf) Writes an array of bytes. |
| void | write(byte[] buf, int off, int len) Writes a sub array of bytes. |
| void | write(int val) Writes a byte. |



| | |
|---|---|
| void | writeChar(int val) Writes a 16 bit char. |
| void | writeChars(String str) Writes a String as a sequence of chars. |
| protected void | writeClassDescriptor(ObjectStreamClass desc) Write the specified class descriptor to th |
| void | writeDouble(double val) Writes a 64 bit double. |
| void | writeFields() Write the buffered fields to the stream. |
| void | writeFloat(float val) Writes a 32 bit float. |
| void | writeInt(int val) Writes a 32 bit int. |
| void | writeLong(long val) Writes a 64 bit long. |
| void | writeObject(Object obj) Write the specified object to the ObjectC |
| protected void | writeObjectOverride(Object obj) Method used by subclasses to override |
| void | writeShort(int val) Writes a 16 bit short |
| protected void | writeStreamHeader() The writeStreamHeader method is provide |
| void | writeUnshared(Object obj) Writes an "unshared" object to the Objec |
| void | writeUTF(String str) Primitive data write of this String in modifi |

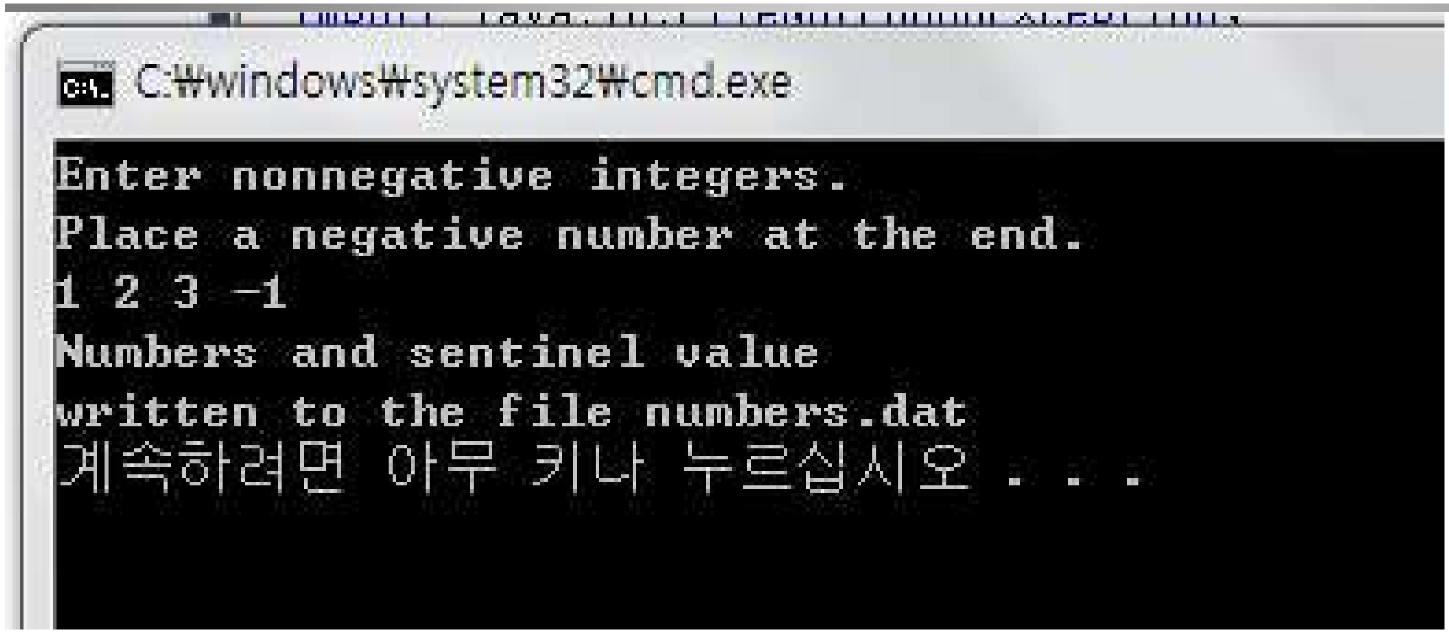
## Listing 10.5 Using ObjectOutputStream to Write to a File - BinaryOutputDemo.java

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

public class BinaryOutputDemo
{
  public static void main(String[] args)
  {
            String fileName = "numbers.dat";
     try
     {
       ObjectOutputStream outputStream =
          new ObjectOutputStream(new FileOutputStream(fileName));
//           new ObjectOutputStream(new File(fileName));
      Scanner keyboard = new Scanner(System.in);
       System.out.println("Enter nonnegative integers.");
       System.out.println("Place a negative number at the end.");

       int anInteger;
do
        {
           anInteger = keyboard.nextInt( );
           outputStream.writeInt(anInteger);
        } while (anInteger >= 0);
        System.out.println("Numbers and sentinel value");
        System.out.println("written to the file " + fileName);
        outputStream.close( );
     }
     catch(FileNotFoundException e)
     {
        System.out.println("Problem opening the file " + fileName);
     }
     catch(IOException e)
     {
        System.out.println("Problem with output to file " + fileName);
     }
   }
}

// public void writeInt(int val)    throws IOException
// public FileOutputStream(File file)     throws FileNotFoundException
```



C:₩windows₩system32₩cmd.exe

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

## When Using **`ObjectOutputStream`** to Output Data to Files:

- **The output files are binary** and can store any of the primitive data types (`int`, `char`, `double`, etc.) and the `String` type
- The files created can be read by other Java programs but are not printable
- The Java I/O library must be imported by including the line: `import java.io.*;`
  - » it contains `ObjectOutputStream` and other useful class definitions
- An `IOException` might be thrown

## Handling **`IOException`**

- `IOException` cannot be ignored
  - » either handle it with a catch block
  - » or defer it with a `throws`-clause

We will put code to open the file and write to it in a `try`-block and write a `catch`-block for this exception :

```
catch(IOException e)
{
  System.out.println("Problem with output...";
}
```

## Opening a New Output File

- The file name is given as a `String`
  - » file name rules are determined by your operating system

- Opening an output file takes two steps
  1. Create a `FileOutputStream` object associated with the file name `String`
  2. Connect the `FileOutputStream` to an `ObjectOutputStream` object

  This can be done in one line of code

# Example: Opening an Output File

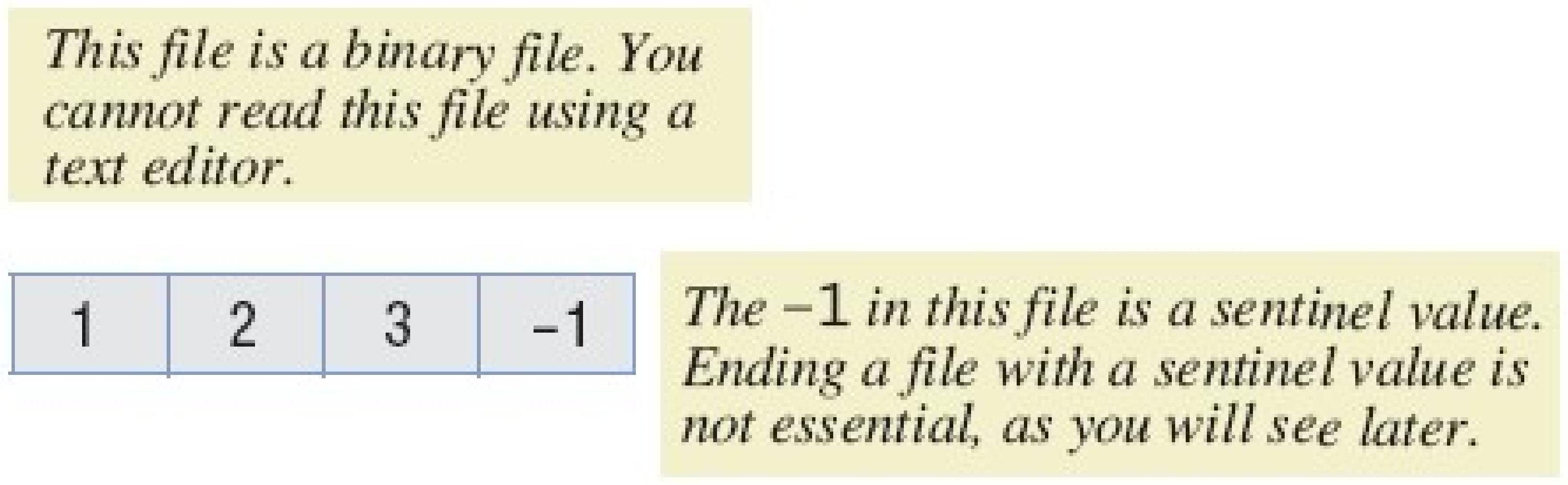
To open a file named `numbers.dat`:

```
ObjectOutputStream outputStream =
  new ObjectOutputStream(
  new FileOutputStream("numbers.dat"));
```

- The constructor for `ObjectOutputStream` requires a `FileOutputStream` argument
- The constructor for `FileOutputStream` requires a `String` argument
  - » the `String` argument is the output file name
- The following two statements are equivalent to the single statement above:

```
 FileOutputStream middleman =
  new FileOutputStream("numbers.dat");
ObjectOutputStream outputStream =
  new ObjectOutputSteam(middleman);
```

# Listing 10.4 Using ObjectOutputStream to Write to a File



*This file is a binary file. You cannot read this file using a text editor.*

*The* `-1` *in this file is a sentinel value. Ending a file with a sentinel value is not essential, as you will see later.*

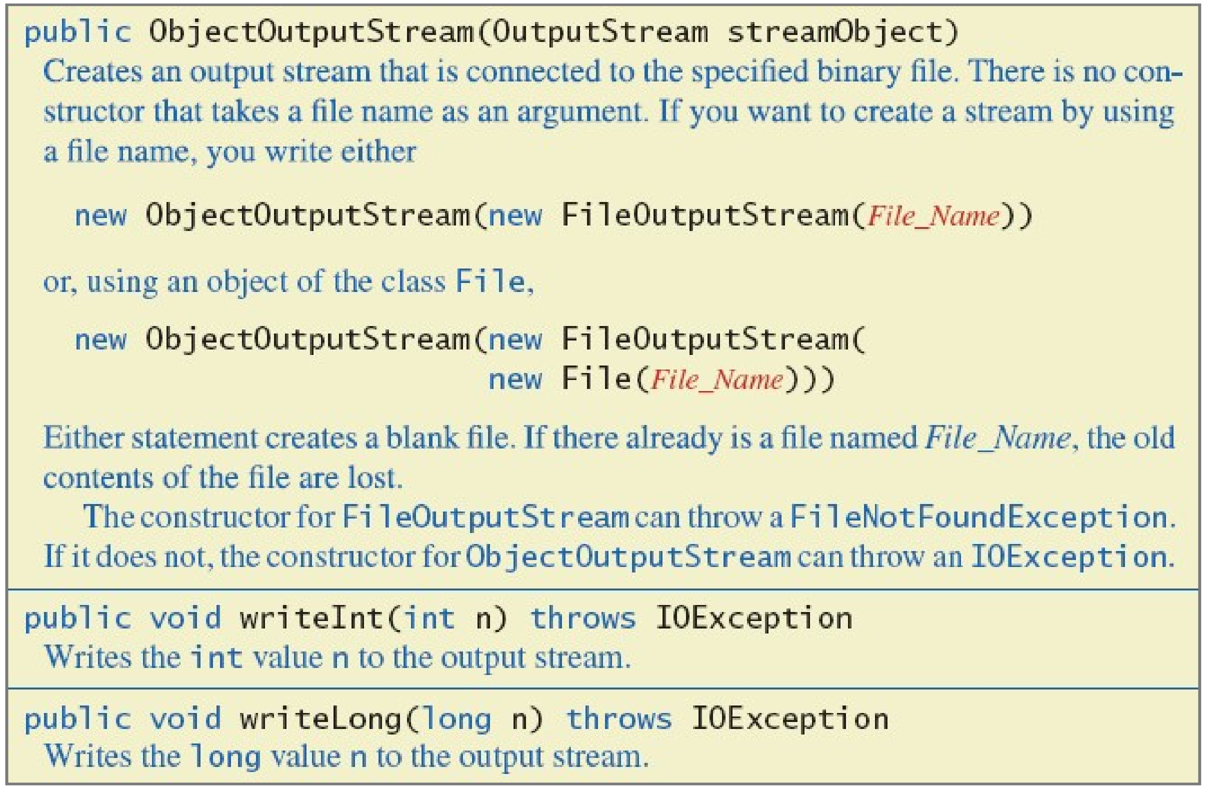
## Some **`ObjectOutputStream`** Methods

- You can write data to an output file after it is connected to a stream class
  - » Use methods defined in `ObjectOutputStream`
    - – `writeInt(int n): write the int value n to the output stream.`
    - – `writeDouble(double x)`
    - – `writeBoolean(boolean b)`
    - – etc.
    - – See the text for more

- Note that each write method throws `IOException`
  - » eventually we will have to write a catch block for it

# Writing Primitive Values to a Binary File

- Figure 10.5a 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`

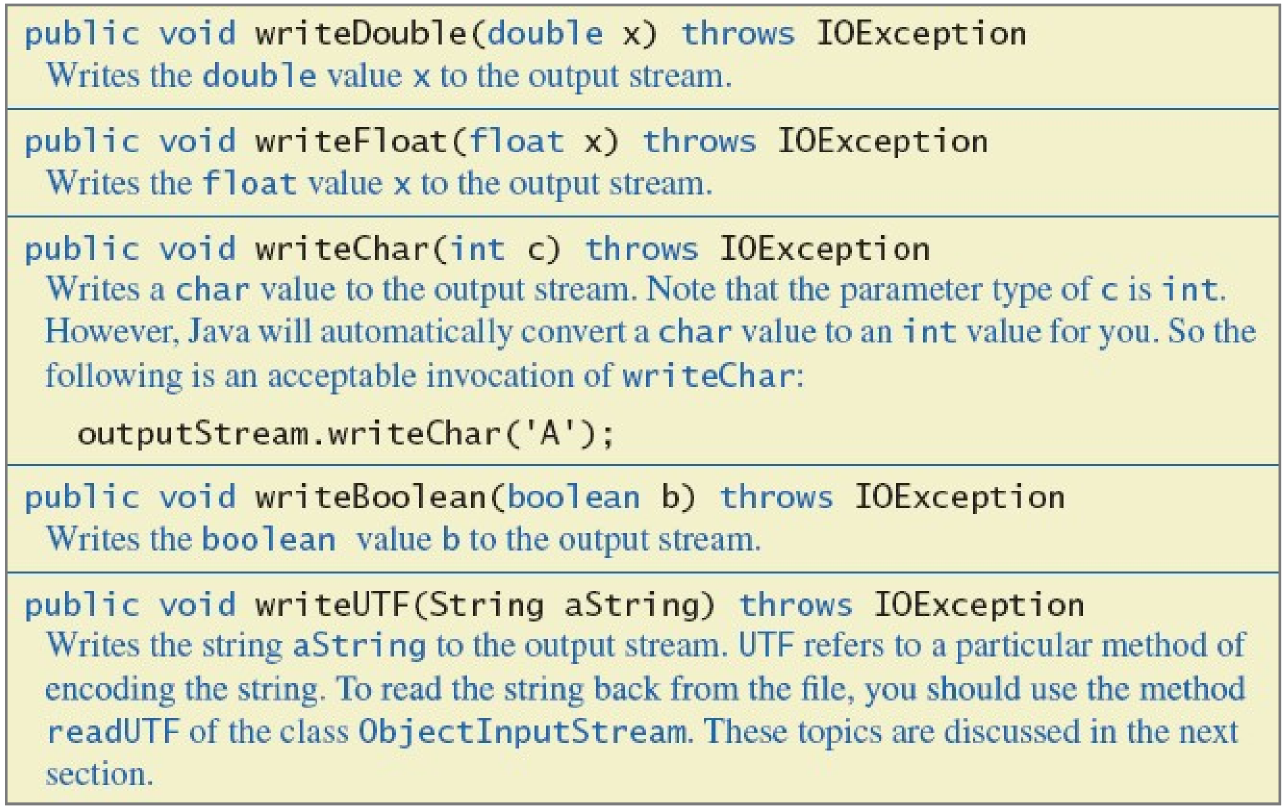
Writes the `int` value `n` to the output stream.

`public void writeLong(long n) throws IOException`

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

# Writing Primitive Values to a Binary File

- Figure 10.5b  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`:

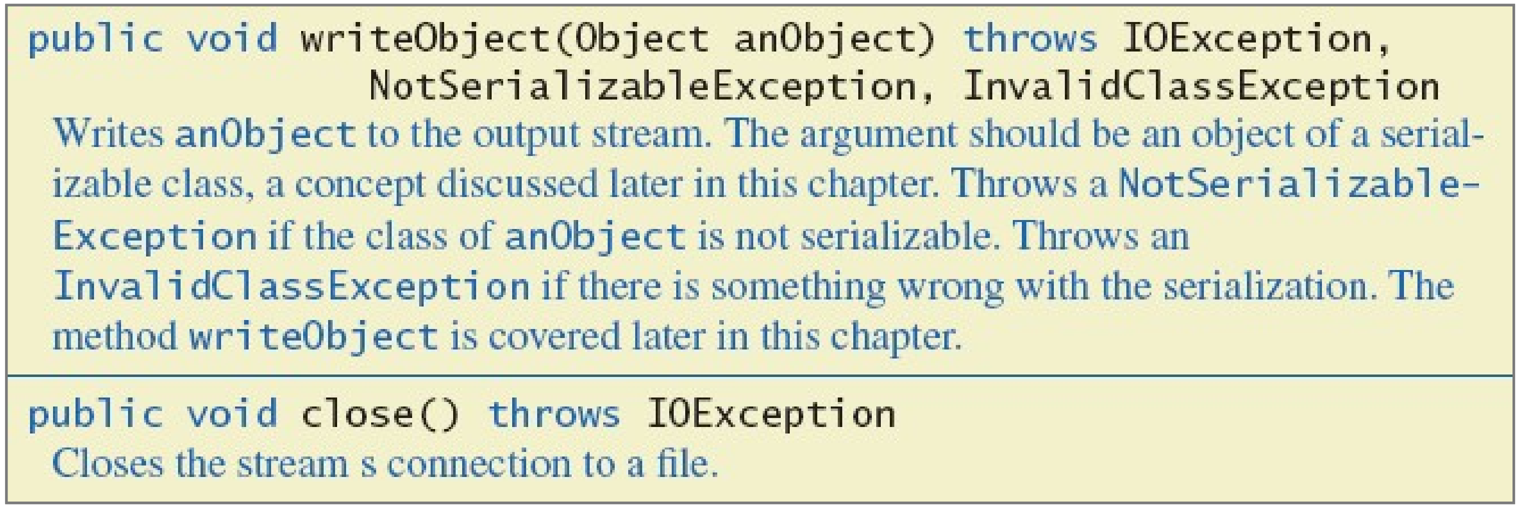
```
outputStream.writeChar('A');
```

`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

- Figure 10.5c  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.

## Closing a File

- An Output file should be closed when you are done writing to it
- Use the `close` method of the class `ObjectOutputStream`
- For example, to close the file opened in the previous example:

```
outputStream.close();
```

- If a program ends normally it will close any files that are open

# Writing a Character to a File: an Unexpected Little Complexity

- The method `writeChar` has an annoying property:
  - » it takes an `int`, not a `char`, argument
- But it is easy to fix:
  - » just cast the character to an int
- For example, to write the character 'A' to the file opened previously:

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

- Or, just use the automatic conversion from `char` to `int`

## Writing a **`boolean`** Value to a File

- `boolean` values can be either of two values, `true` or `false`
- `true` and `false` are not just names for the values, they actually are of type `boolean`
- For example, to write the `boolean` value `false` to the output file:

```
outputStream.writeBoolean(false);
```

## Writing Strings to a File: Another Little Unexpected Complexity

- Use the `writeUTF` method to output a value of type `String`
  - » there is no `writeString` method
- UTF stands for Unicode Text Format
  - » a special version of Unicode
- Unicode: a text (printable) code that uses 2 bytes per character
  - » designed to accommodate languages with a different alphabet or no alphabet (such as Chinese and Japanese)
- ASCII: also a text (printable) code, but it uses just 1 byte per character
  - » the most common code for English and languages with a similar alphabet
- UTF is a modification of Unicode that uses just one byte for ASCII characters
  - » allows other languages without sacrificing efficiency for ASCII files

## When Using `ObjectInputStream` to Read Data from Files

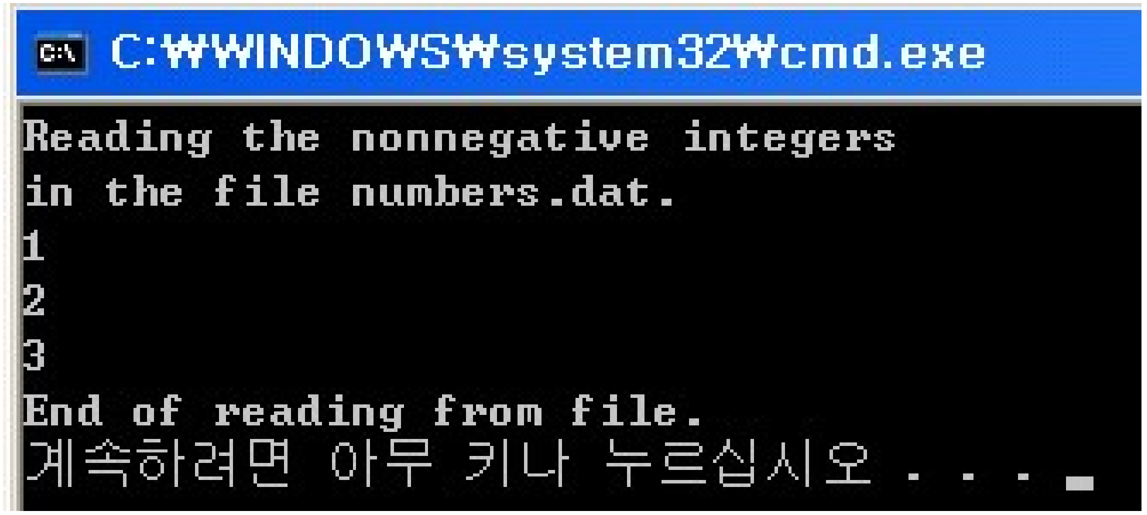
- Input files are binary and contain any of the primitive data types (`int`, `char`, `double`, etc.) and the `String` type
- The files can be read by Java programs but are [redacted]
- The Java I/O library must be imported including the line:
  `import java.io.*;`
  - » it contains `ObjectInputStream` and other useful class definitions
- An `IOException` might be thrown

# Listing 10.6 Using ObjectInputStream to read from a File - BinaryInputDemo.java

» Notice that the sentinel value –1 is read from the file but is not output to the screen.

```
 import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

public class BinaryInputDemo
{
  public static void main(String[] args)
  {
     String fileName = "numbers.dat";
     try
     {
        ObjectInputStream inputStream =
             new ObjectInputStream(new FileInputStream(fileName));
        System.out.println("Reading the nonnegative integers");
        System.out.println("in the file " + fileName);
        int anInteger = inputStream.readInt( );
        while (anInteger >= 0)
        {
            System.out.println(anInteger);
            anInteger = inputStream.readInt( );
        }
        System.out.println("End of reading from file.");
        inputStream.close( );
    }
    catch(FileNotFoundException e)
    {
        System.out.println("Problem opening the file " + fileName);
    }
    catch(EOFException e)
    {
        System.out.println("Problem reading the file " + fileName);
        System.out.println("Reached end of the file.");
    }
    catch(IOException e)
    {
        System.out.println("Problem reading the file " + fileName);
    }
  } // public int readInt() throws IOException, EOFException
} //public FileInputStream(File file) throws FileNotFoundException
```



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

# Opening a New Input File

- Similar to opening an output file, but replace "output" with "input"
- The file name is given as a `String`
  - » file name rules are determined by your operating system
- Opening a file takes two steps
  1. Creating a `FileInputStream` object associated with the file name `String`
  2. Connecting the `FileInputStream` to an `ObjectInputStream` object
- This can be done in one line of code

## Example: Opening an Input File

To open a file named `numbers.dat`:

```
ObjectInputStream inStream =
  new ObjectInputStream (new
  FileInputStream("numbers.dat"));
```

- The constructor for `ObjectInputStream` requires a `FileInputStream` argument
- The constructor for `FileInputStream` requires a `String` argument
  - » the `String` argument is the input file name
- The following two statements are equivalent to the statement at the top of this slide:

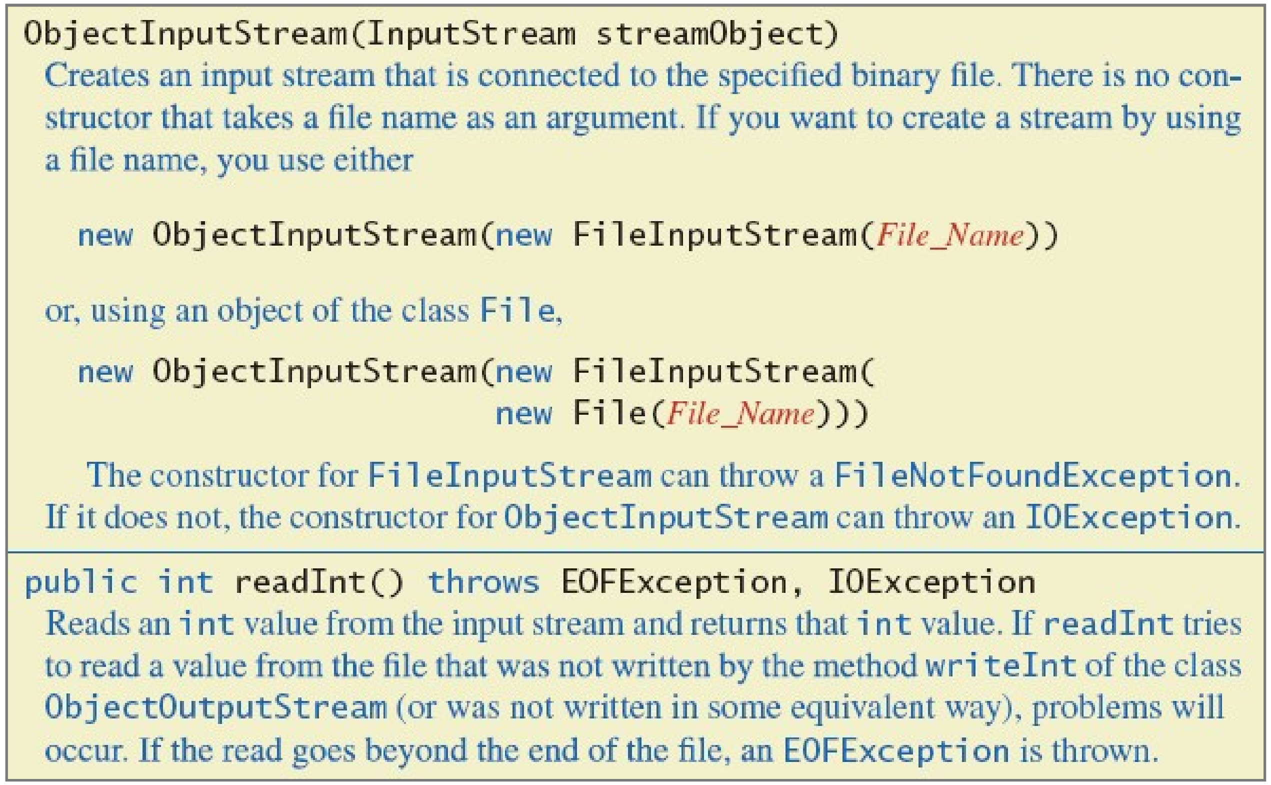
```
FileInputStream middleman =
  new FileInputStream("numbers.dat");
ObjectInputStream inputStream =
  new ObjectInputStream (middleman);
```

## Some **`ObjectInputStream`** Methods

- For every output file method there is a corresponding input file method
- You can read data from an input file after it is connected to a stream class
  - » Use methods defined in `ObjectInputStream`
    - – `readInt()`
    - – `readDouble()`
    - – `readBoolean()`
    - – etc.
    - – See the text for more
- Note that each write method throws `IOException`

## Reading from a Binary File

- Figure 10.6a  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)))
```

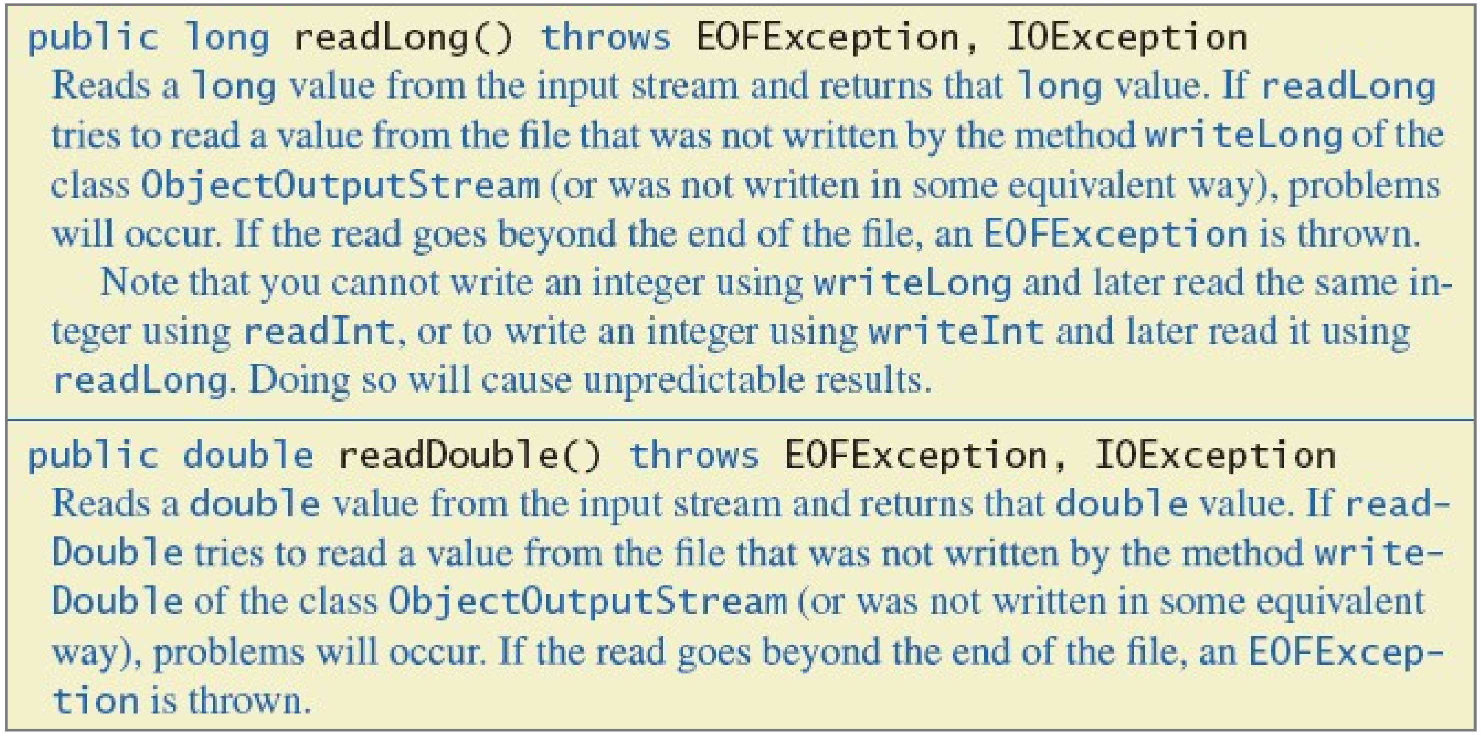
The constructor for `FileInputStream` can throw a `FileNotFoundException`. If it does not, the constructor for `ObjectInputStream` can throw an `IOException`.

`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

- Figure 10.6b  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.

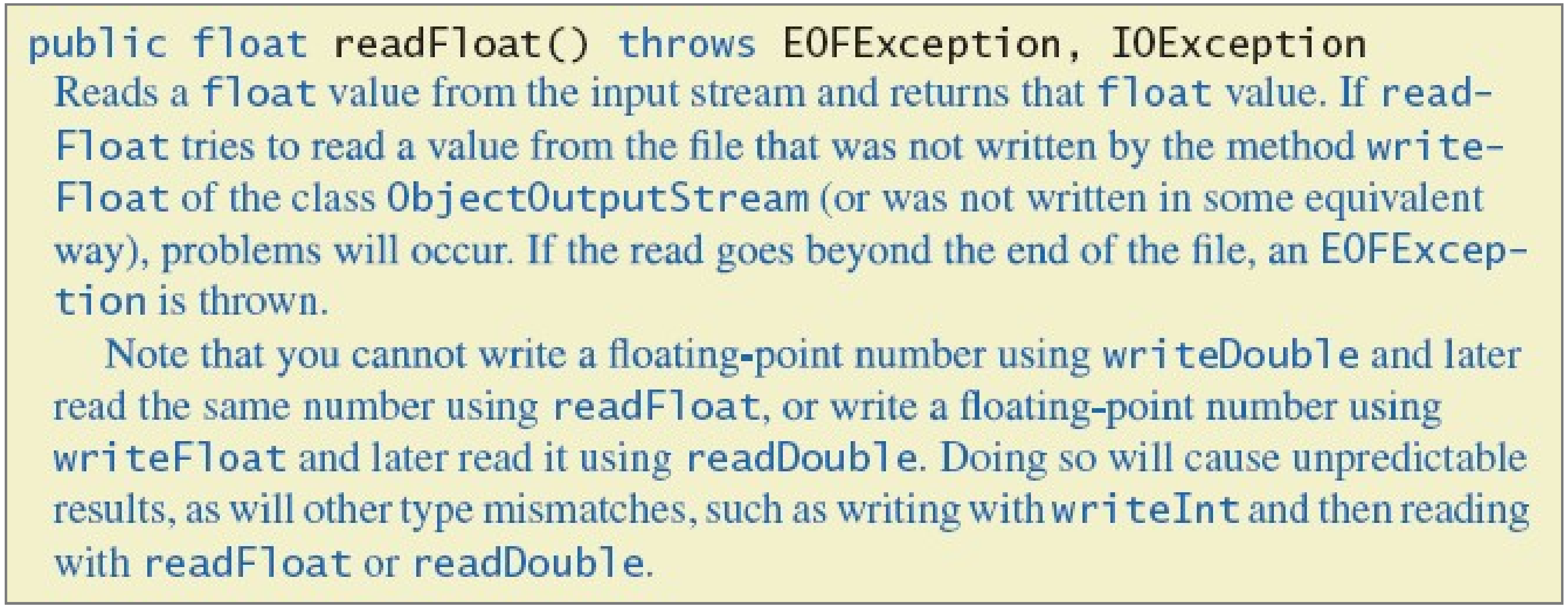
Note that you cannot write an integer using `writeLong` and later read the same integer using `readInt`, or to write an integer using `writeInt` and later read it using `readLong`. Doing so will cause unpredictable results.

`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

- Figure 10.6c  Some methods of class **`ObjectInputStream`**



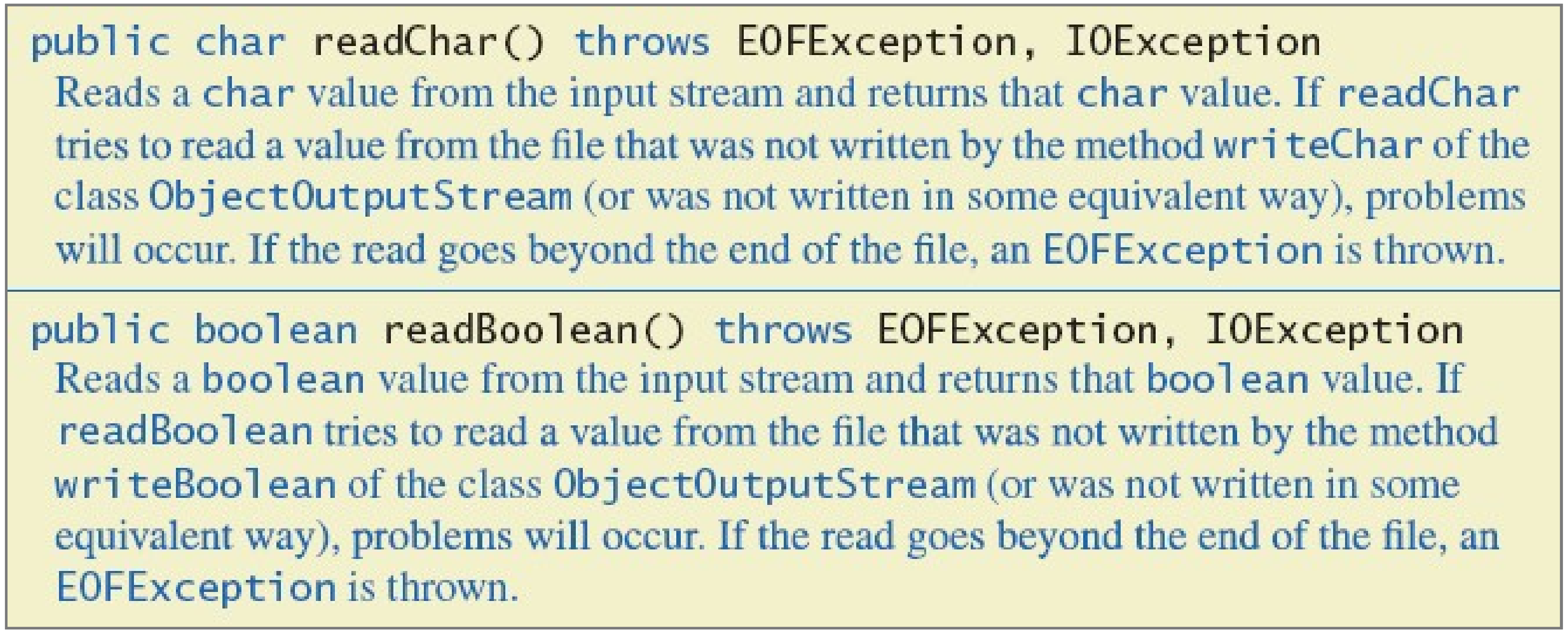
`public float readFloat() throws EOFException, IOException`

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

- Figure 10.6d Some methods of class **`ObjectInputStream`**



`public char readChar() throws EOFException, IOException`

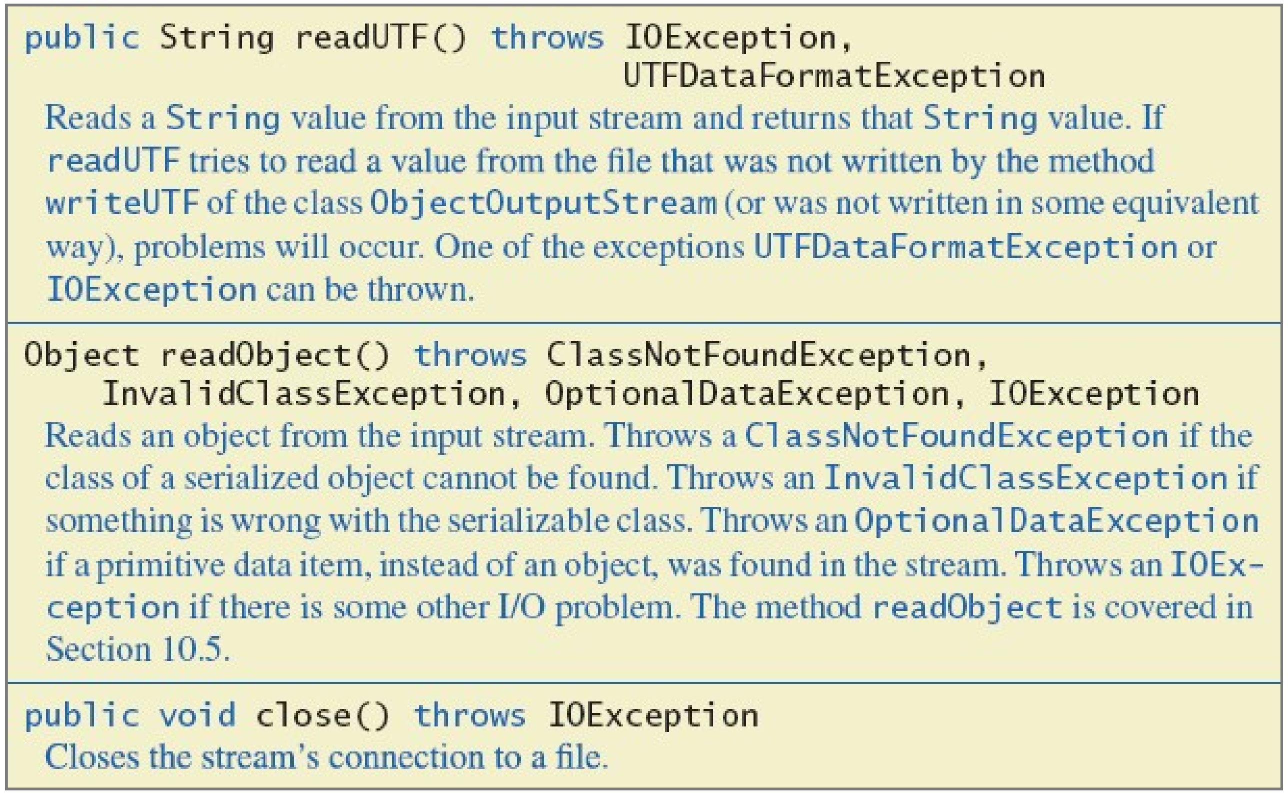
Reads a `char` value from the input stream and returns that `char` value. If `readChar` tries to read a value from the file that was not written by the method `writeChar` of the class `ObjectOutputStream` (or was not written in some equivalent way), problems will occur. If the read goes beyond the end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

- Figure 10.6e 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.

## Input File Exceptions

- A `FileNotFoundException` is thrown if the file is not found when an attempt is made to open a file
- Each read method throws `IOException`
  - » we still have to write a catch block for it
- If a read goes beyond the end of the file an `EOFException` is thrown

## Avoiding Common **`ObjectInputStream`** File Errors

There is no error message (or exception)
if you read the wrong data type!

- Input files can contain a mix of data types
  - » it is up to the programmer to know their order and use the correct read method

- `ObjectInputStream` works with binary, not text files

- As with an output file, close the input file when you are done with it

## Common Methods to Test for the End of an Input File

- A common programming situation is to read data from an input file but not know how much data the file contains
- In these situations you need to check for the end of the file
- There are three common ways to test for the end of a file:
  1. Put a sentinel value at the end of the file and test for it.
  2. Throw and catch an end-of-file exception.
  3. Test for a special character that signals the end of the file (text files often have such a character).

## The **`EOFException`** Class

- Many (but not all) methods that read from a file throw an end-of-file exception (`EOFException`) when they try to read beyond the file
  - » all the `ObjectInputStream` methods in Display 9.3 do throw it
- The end-of-file exception can be used in an "infinite" (`while(true)`) loop that reads and processes data from the file
  - » the loop terminates when an `EOFException` is thrown
- The program is written to continue normally after the `EOFException` has been caught

## Using **EOFException**

`main` method from `EOFExceptionDemo`

Intentional "infinite" loop to process data from input file

Loop exits when end-of-file exception is thrown

Processing continues after `EOFException`: the input file is closed

Note order of catch blocks: the most specific is first and the most general last

```
try
{
   ObjectInputStream inputStream =
     new ObjectInputStream(new FileInputStream("numbers.dat"));
   int n;

   System.out.println("Reading ALL the integers");
   System.out.println("in the file numbers.dat.");
   try
   {
      while (true)
      {
         n = inputStream.readInt();
         System.out.println(n);
      }
   }
   catch(EOFException e)
   {
      System.out.println("End of reading from file.");
   }
   inputStream.close();
}
catch(FileNotFoundException e)
{
   System.out.println("Cannot find file numbers.dat.");
}
catch(IOException e)
{
   System.out.println("Problem with input from file numbers.dat.");
}
```

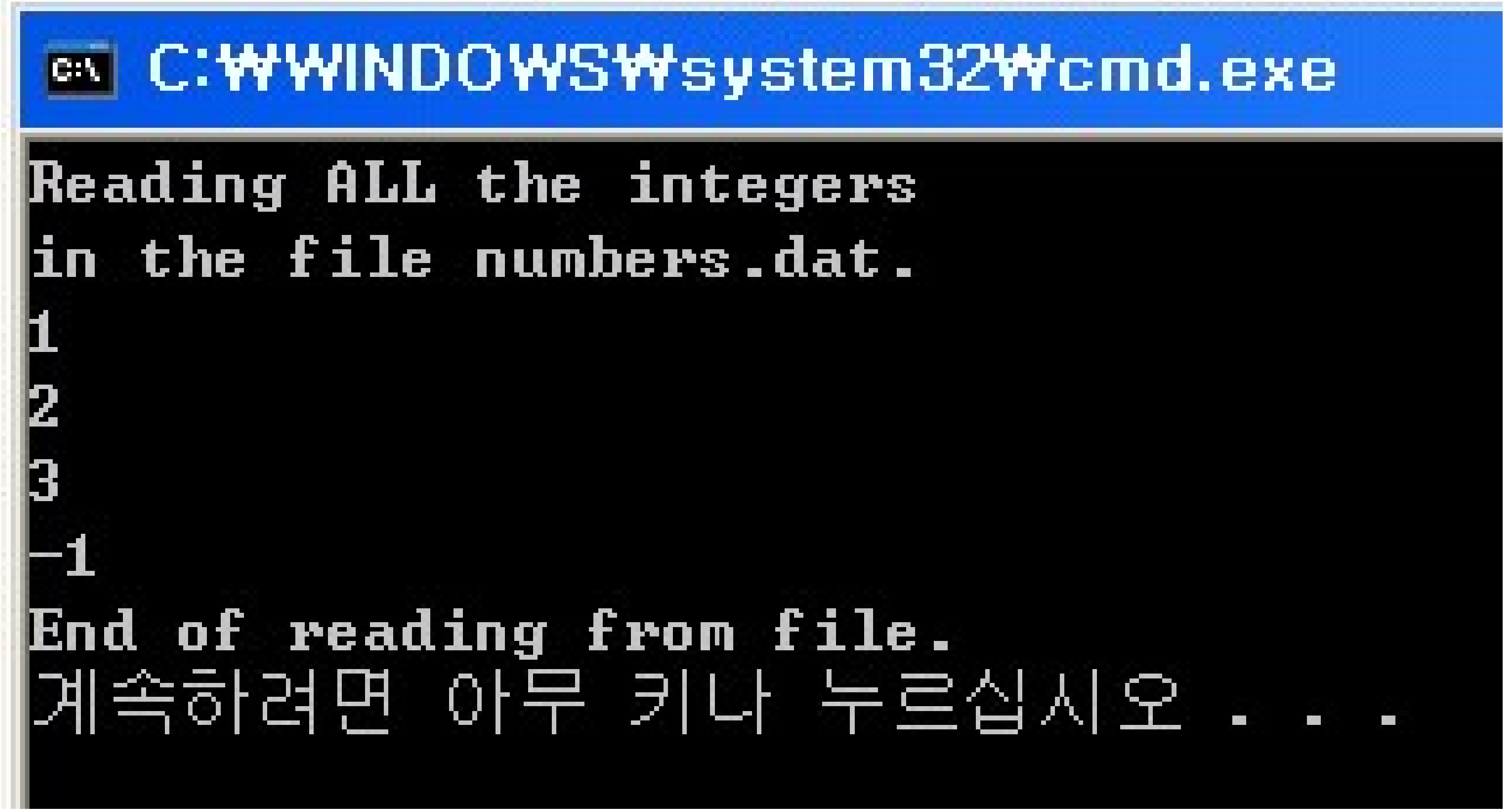
## Listing 10.7 Using EOFException - EOFExceptionDemo.java

- Notice that, when you use EOFException, -1 is treated just like any other integer. EOFException allows you to have files that contain any kind of integers.

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

public class EOFExceptionDemo
{
   public static void main(String[] args)
   {
      String fileName = "numbers.dat";
      try
      {
         ObjectInputStream inputStream =
             new ObjectInputStream(new FileInputStream(fileName));
         System.out.println("Reading ALL the integers");
         System.out.println("in the file " + fileName);
```

```
            try
            {
               while (true)
               {
                  int anInteger = inputStream.readInt( );
                  System.out.println(anInteger);
               }
            }
            catch(EOFException e)
            {
               System.out.println("End of reading from file.");
            }
            inputStream.close( );
         }
         catch(FileNotFoundException e)
         {
            System.out.println("Cannot find file " + fileName);
         }
         catch(IOException e)
         {
            System.out.println("Problem with input from file " + fileName);
         }
      }
   }
```



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

## Listing 10.8 Processing a File of Binary Data - Doubler.java

- asks the user for the two file names
  - then, copies all the numbers in one file into the other file
  - Multiples each number by two

```
 //Listing 10.8  Processing a File of Binary Data

import java.io.*;

public class Doubler
{
   private ObjectInputStream inputStream = null;
   private ObjectOutputStream outputStream = null;
```

```
/**
  Doubles the integers in one file and puts them in another file.
 */
 public static void main(String[] args)
 {
     Doubler twoTimer = new Doubler( );
     twoTimer.connectToInputFile( );
     twoTimer.connectToOutputFile( );
     twoTimer.timesTwo( );
     twoTimer.closeFiles( );
     System.out.println("Numbers from input file");
     System.out.println("doubled and copied to output file.");
 }
```

```
public void connectToOutputFile( )
{
    String outputFileName =
                getFileName("Enter output file name:");
    try
    {
        outputStream = new ObjectOutputStream(
                new FileOutputStream(outputFileName));
    }
    catch(IOException e)
    {
        System.out.println("Error opening output file "
                                + outputFileName);
        System.out.println(e.getMessage( ));
        System.exit(0);
    }
}
```

```
public void connectToInputFile( )
   {
       String inputFileName =
                getFileName("Enter input file name:");

       try
       {
          inputStream =
            new ObjectInputStream(
                   new FileInputStream(inputFileName));
       }
       catch(FileNotFoundException e)
       {
          System.out.println("File " + inputFileName
                               + " not found.");
          System.exit(0);
       }
       catch(IOException e)
       {
          System.out.println("Error opening input file "
                                   + inputFileName);
          System.exit(0);
       }
   }
```

```
private String getFileName(String prompt)
{
    String fileName = null;
    System.out.println(prompt);
    Scanner keyboard = new Scanner(System.in);
    fileName = keyboard.next( );
    return fileName;
}

public void timesTwo( )
{
    int next;
    try
    {
        while (true)
        {
            next = inputStream.readInt( );
            outputStream.writeInt(2*next);
        }
    }
    catch(EOFException e)
    {
        //Do nothing. This just ends the loop.
    }
```

```
 catch(IOException e)
        {
            System.out.println(
                   "Error: reading or writing files.");
            System.out.println(e.getMessage( ));
            System.exit(0);
        }
    }


    public void closeFiles( )
    {
        try
        {
            inputStream.close( );
            outputStream.close( );
        }
        catch(IOException e)
        {
            System.out.println("Error closing files "
                                 + e.getMessage( ));
            System.exit(0);
        }
    }
}
```

끝