CS 112 – MiraCosta College

Introduction to Computer Science II
Java

Module 6 – Binary Files

Chris Merrill
Computer Science Dept
cmerrill@miracosta.edu

Agenda

- Review of last Module
- Chapter 10 (continued)
  - The File Class
  - Binary Files
  - Writing Objects
- Lab – Writing binary and objects to 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
  - The `System.in` object 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
  - The `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* if the data they contain uses an ASCII encoding scheme
- An advantage of text files is that the 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

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`**, **`println`**, and **`printf`**
- 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

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

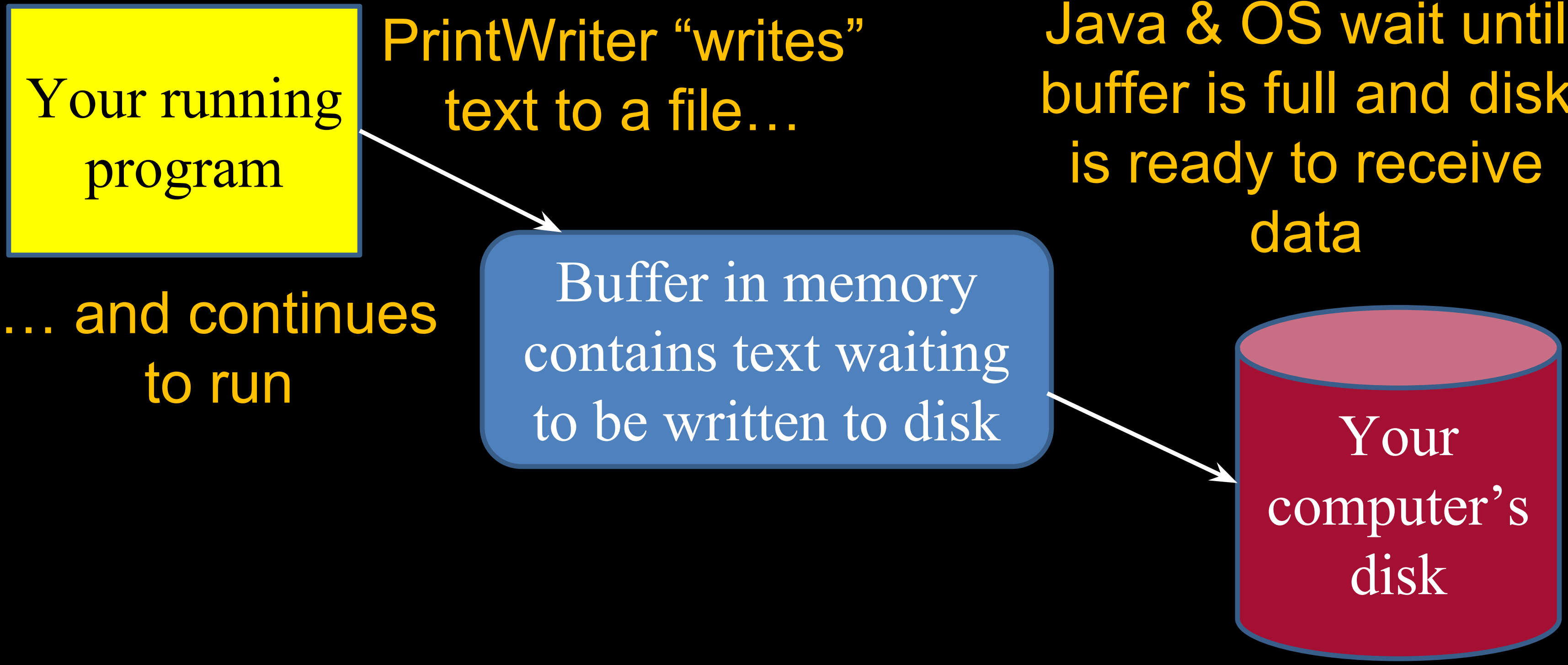
- If the file already exists, then doing this causes the old contents to be lost (or use the append version on the prior slide).
- If the file does not exist, then a new, empty file named ***`FileName`*** is created
- After doing this, the methods **`print`**, **`println`**, and **`printf`** can be used to write to the file

File Buffers and the `flush` Method

Output streams connected to files are usually *buffered*

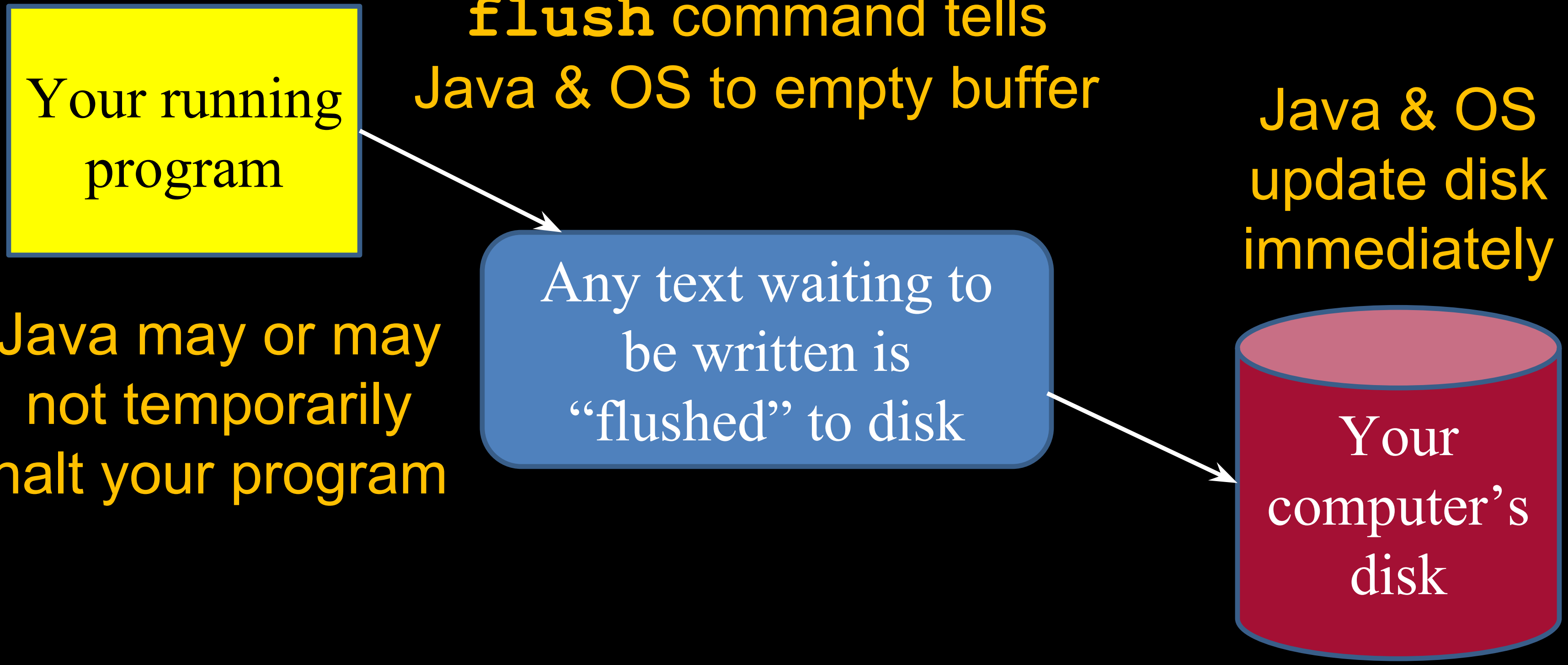
- Rather than physically writing to the file as soon as possible, the data is saved in a temporary location (*buffer*)
- When enough data accumulates, or when the method `flush` is invoked, the buffered data is written to the file all at once
- This is more efficient, since physical writes to a file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

`PrintWriter` Output Buffer


Your running program
PrintWriter "writes" text to a file…
… and continues to run
Buffer in memory contains text waiting to be written to disk
Java & OS wait until buffer is full and disk is ready to receive data
Your computer's disk

`flush` empties buffer

`flush` command tells Java & OS to empty buffer



Java may or may not temporarily halt your program

Java & OS update disk immediately

`close` invokes `flush` automatically

The method `close` invokes the method `flush`, thus insuring that all the data is written to the file

- If a program relies on Java to close the file, and the program terminates abnormally (or just stops), then any output that was buffered may not get written to the file
- The sooner a file is closed after writing to it, the less likely it is that there will be a problem

Reading From a Text File Using `Scanner`

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));
```

Using a `String` as a Parameter to the Constructor

The `Scanner` also has a constructor that takes a `String` as a parameter.

- Unfortunately, this not treated as a file name
- Instead, its treated as a `String` object to be scanned.

Reading From a Text File Using **`Scanner`**

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`**, **`nextDouble`**, **`next`**, and **`nextLine`** methods

“Testing” Methods in the `Scanner` class

- A program that tries to read beyond the end of a file using methods of the `Scanner` class will cause an exception 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 a Text File Using **`BufferedReader`**

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

```
BufferedReader readerObject;
readerObject =
     new BufferedReader(
          new FileReader(FileName));
```

This opens the file for reading

Reading From a Text File

After opening the file, the methods `read` and `readLine` can be used to read from the file

- The `readLine` method is the same method used to read from the keyboard, but in this case it would read from a file
- The `read` method reads a single character, and returns a value (of type `int`) that corresponds to the character read
- Since the read method does not return the character itself, a type cast must be used:

```
char next = (char)(readerObject.read());
```

Reading Numbers

Unlike the **`Scanner`** class, the **`BufferedReader`** class has no methods to read a number from a text file

- Instead, a number must be read in as a string, and then converted to a value of the appropriate numeric type using one of the wrapper classes
- To read in a single number on a line by itself, first use the method **`readLine`**, and then convert the string into a number

Reading Numbers

- Use the wrapper methods **`Integer.parseInt`**, **`Double.parseDouble`**, etc. to convert the string into a number
- If there are multiple numbers on a line, **`StringTokenizer`** can be used to decompose the string into tokens, and then the tokens can be converted as described above

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 program is located

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

Some Methods in the `File` Class

`File` is in the `java.io` package

Constructor:

```
        public File(String filename)
```

The ***`filename`*** can contain either a full or a relative path name.

`filename` is referred to as the *abstract path*.

Some Methods in the **`File`** Class

`public boolean exists()`

Returns **`true`** if the file already exists, otherwise returns **`false`**.

`public boolean canRead()`

Tests whether the program can read from the file. Returns **`true`** if the file exists and can be read, otherwise returns **`false`**

Some Methods in the **`File`** Class

`public boolean canWrite()`

Tests whether the program can write tothe file. Returns **`true`** if the file exists and is writeable, otherwise returns **`false`**

`public boolean setReadOnly()`

Sets the file to be readable only. Returns **`true`** if the successful, otherwise returns **`false`**

Some Methods in the **`File`** Class

```
public boolean delete()
```

Tries to delete the file or directory (folder) named by the abstract path. (A directory must be empty to be deleted.)

Returns **`true`** if the file exists and is writeable, otherwise returns **`false`**

```
public boolean createNewFile()
                                throws IOException
```

Creates a new file named by the abstract path, provided the file does not already exist. Returns **`true`** if the successful, otherwise returns **`false`**

Some Methods in the **`File`** Class

`public String getName()`

Returns the abstract path name as a String.

`public boolean renameTo(File *newName*)`

Renames the file represented by the abstract path ame to ***`newName`***. ***`newName`*** can be either a relative or full path name, which may require moving the file. Returns **`true`** if the rename operation was successful, otherwise returns **`false`**

Some Methods in the **`File`** Class

`public boolean isFile()`

Tests if a file exists that is named by the abstract path. Returns **`true`** if the file exists, otherwise returns **`false`**.

`public boolean isDirectory()`

Test if the directory (folder) exists that is named by the abstract path. Returns **`true`** if the directory exists, otherwise returns **`false`**.

Some Methods in the `File` Class

`public boolean mkDir()`

Makes a directory named by the abstract path. (Will *not* create parent directories.) Returns **`true`** if directory was created, otherwise returns **`false`**.

`public long length()`

Returns the length (in bytes) of the file named by the abstract path. The return value is *unspecified* if the file does not exist.

Other Methods in the Class **`File`**

```
public String getAbsolutePath()
public String getParent()
public boolean canRead()
public long lastModified()
public void setLastModified(long)
public String[] list()    (lists files in folder)
public boolean equals(File)
public String toString()
```

Demonstration of the **`File`** Class

Demonstration of deleting, creating, and setting attributes of a file using methods in the **`File`** class.

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

- The constructor for `FileOutputStream` may throw a `FileNotFoundException`
- The constructor for `ObjectOutputStream` may throw an `IOException`

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`

Methods in the Class `ObjectOutputStream`

`ObjectOutputStream` and `FileOutputStream` are in the `java.io` package

Constructor:

```
public ObjectOutputStream(
              FileOutputStream outputStream)
```

To create on disk:

```
new ObjectOutputStream(
         new FileOutputStream(filename))
```

This will create an empty file with the new ***`filename`***.

Methods in the Class `ObjectOutputStream`

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

Writes the `int` value of n into the output stream.

```
public void writeShort(short n)
              throws IOException
```

Writes the `short` value of n into the output stream.

```
public void writeDouble(double x)
              throws IOException
```

Writes the `double` value of x into the output stream.

Methods in the Class `ObjectOutputStream`

```
public void writeFloat(float x)
                 throws IOException
```

Writes the **`float`** value of x into the output stream.

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

Writes the **`char`** value of n into the output stream. Note that this method takes an **`int`** as an argument. If a **`char`** is provided, java will typecast it to an **`int`** automatically.

Methods in the Class `ObjectOutputStream`

```
public void writeBoolean (boolean b)
                  throws IOException
```

Writes the **`boolean`** value of b into the output stream.

```
public void writeUTF(String aString)
                  throws IOException
```

Writes the **`String`** value of **`aString`** into the output stream. UTF refers to a particular method of encoding characters. Use **`readUTF`** to read the **`String`** back from the file.

Methods in the Class `ObjectOutputStream`

```
public void writeObject(Object anObject)
                        throws IOException
```

Writes the `Object` given as the argument into the output stream.

- The object should be `Serializable`.
- Child objects or other objects assigned to instance variables must also be `Serializable`.

Methods in the Class **`ObjectOutputStream`**

`public void close() throws IOException`

Closes the stream's connection to a file. This method invokes **`flush()`** first.

`public void flush() throws IOException`

Flushes the output stream. This forces an actual write to the file of any data that has been buffered and not yet physically written. (Normally you do not need to use this method.

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**
- Both of these exceptions *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 read 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

Methods in the Class `ObjectInputStream`

`ObjectInputStream` and **`FileInputStream`** are in the **`java.io`** package

Constructor:

```
public ObjectInputStream(
         FileInputStream inputStream)
```

To read from an existing on disk:

```
new ObjectInputStream(
         new FileInputStream(filename))
```

Methods in the Class `ObjectInputStream`

Alternately, you can use a **`File`** object to read from an existing on disk:

```
new ObjectInputStream(
         new FileInputStream(fileObject))
```

Both versions can throw a **`FileNotFoundException`** (if this file doesn't exit) or an **`IOException`**.

Methods in the Class **`ObjectInputStream`**

`public void readInt() throws IOException` *

Reads an **`int`** value from the input stream and returns it. The value should have been written using the **`writeInt`** method.

`public void readShort() throws IOException` *

Reads a **`short`** value from the input stream and returns it. The value should have been written using the **`writeShort`** method.

* can throw an **`EOFException`**

Methods in the Class **`ObjectInputStream`**

`public long readLong() throws IOException` *

Reads a **`long`** value from the input stream and returns it. The value should have been written using the **`writeLong`** method.

`public double readDouble() throws IOException`*

Reads a **`double`** value from the input stream and returns it. The value should have been written using the **`writeDouble`** method.

* can throw an **`EOFException`**

Methods in the Class `ObjectInputStream`

`public float readFloat() throws IOException` *

Reads a **`float`** value from the input stream and returns it. The value should have been written using the **`writeFloat`** method.

`public char readChar() throws IOException` *

Reads a **`char`** value from the input stream and returns it. The value should have been written using the **`writeChar`** method.

* can throw an **`EOFException`**

Methods in the Class **`ObjectInputStream`**

`public boolean readBoolean() throws IOException`*

Reads a **`boolean`** value from the input stream and returns it. The value should have been written using the **`writeBoolean`** method.

`public String readUTF() throws IOException` *

Reads a **`String`** value from the input stream and returns it. The value should have been written using the **`writeUTF`** method.

* can throw an **`EOFException`**

Methods in the Class `ObjectInputStream`

```
public Object readObject() throws IOException*
```

Reads an `object` value from the input stream and returns it. The value should have been written using the `writeObject` method.

```
public int skipBytes(int n)
                     throws IOException *
```

Skips n bytes in the input stream

* can throw an `EOFException`

Demonstrate Binary File Access and Updates

Use one program to write binary data (**`int`**, **`byte`**, **`double`**, **`String`**) onto a file, then another program to read that data back in.

Checking for the End of a Binary File

- 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

Binary I/O of Objects

Objects can also be input and output from a binary file

- Use `writeObject` (of the `ObjectOutputStream` class) to write an object to a binary file
- Use `readObject` (of the `ObjectInputStream class`) 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 typecast first:

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

Binary I/O of Objects

- It is best to store the data of only one class type in any one file
  - Storing objects of multiple class types or objects of one class type mixed with primitives can lead to loss of data
- In addition, 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 AClass implements Serializable
```

- When a serializable class has instance variables of a class type, then 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

Array Objects in Binary Files

Since an array is an object, arrays can 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), typecast it to the correct type:

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

Appending to a Binary File

To append objects to an existing binary file, we have to get around a "header" problem:

- The `FileOutputStream` has a constructor with a second parameter which, if set to true, opens an existing file and appends to it properly.
- However, binary files have "headers" which contain information about the Serialization of that file.
- To append to an existing file, we have to get around writing another header.

Appending to a Binary File

- One way to do this is to override the method that writes the header in the `ObjectOutputStream`
  - The method that writes the header to the file is named `writeStreamHeader`
  - We'll replace the body of this method with a call to `reset` instead

Demo Writing and Appending to a Binary File

Create a simple class with one instance variables, read and write objects to that file (including appending)

Random Access to Binary Files

- The streams for sequential access to files are the ones most commonly used for file access in Java
- However, some applications require very rapid access to records in very large databases
  - These applications need to have random access to particular parts of a file

Reading and Writing to the Same File

- The stream class **`RandomAccessFile`**, which is in the **`java.io`** package, provides both read and write random access to a file in Java
- A random access file consists of a sequence of numbered bytes
  - There is a type of marker called the *file pointer* that is always positioned at one of the bytes in the file
  - All reads and writes take place starting at the *file pointer location*
  - The file pointer can be moved to a new location with the method **`seek`**

Reading and Writing to the Same File

Although a random access file is *byte-oriented*, there are methods that allow for reading or writing values of the primitive types as well as string values to/from a random access file

- These include **`readInt`**, **`readDouble`**, and **`readUTF`** for input, and **`writeInt`**, **`writeDouble`**, **`writeUTF`** for output

- However, it does not have a **`writeObject`** or a **`readObject`** method

Opening a Random Access File

- The constructor for **`RandomAccessFile`** takes either a string file name or an object of the class **`File`** as its first argument
- The second argument must be one of four strings:
  - **`"rw"`**, meaning the code can both read and write to the file after it is open
  - **`"r"`**, meaning the code can only read from the file
  - **`"rws"`** or **`"rwd"`** (See Table of methods from **`RandomAccessFile`**)

Random Access Files Need Not Start Empty

- If the file already exists, then when it is opened, the length is not reset to 0, and the file pointer will be positioned at the start of the file
  - This ensures that old data is not lost, and that the file pointer is set for the most likely position for reading (not writing)
- The length of the file can be changed with the `setLength` method
  - In particular, the `setLength` method can be used to empty the file

Methods in the Class **RandomAccessFile**

The class **`RandomAccessFile`** is in the **`java.io`** package

Constructor:

```
public RandomAccessFile(String fileName)
```

To open an existing on disk:

```
new RandomAccessFile(String filename,
                     String mode)

new RandomAccessFile(File fileobject,
                     String mode)
```

Constructor modes

“r”	Open for reading only
“rw”	Open for reading and writing
“rws”	Same as “rw”, but physical write is done at same time for file content or metadata
“rwd”	Same as “rw”, but physical write is done at same time for file content.

Methods in the Class **`RandomAccessFile`**

```
public long getFilePointer()
                         throws IOException
```

Returns the current location of the *file pointer*. Locations are numbered starting at 0.

```
public void seek(long location)
                         throws IOException
```

Moves the *file pointer* to the specified location.

Methods in the Class **`RandomAccessFile`**

```
public long length() throws IOException
```

Returns the length (size) of the file.

```
public void setLength(long newLength)
                         throws IOException
```

Sets the length of the file.

- If the present length is greater than the new length, then the file will be truncated.
- If the present length is less than the new length, the file will be extended (with the new content undefined).

Methods in the Class **`RandomAccessFile`**

`public long close() throws IOException`

Closes the stream's connection to the file

```
public final void writeByte(int b)
                     throws IOException
```

Writes the specified byte value to the file

```
public void writeByte(byte[] a)
                     throws IOException
```

Writes **`a.length`** bytes to the specified file

Methods in the Class **`RandomAccessFile`**

```
public final void writeShort(short s)
                                    throws IOException
```

Writes the short **`s`** to the file

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

Writes the integer **`n`** to the file

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

Writes the long **`n`** to the file

Methods in the Class **`RandomAccessFile`**

```
public final void writeFloat(float f)
                                   throws IOException
```

Writes the float `f` to the file

```
public final void writeDouble(double d)
                                   throws IOException
```

Writes the double `d` to the file

```
public final void writeChar(char c)
                                   throws IOException
```

Writes the char `c` to the file

Methods in the Class **`RandomAccessFile`**

```
public final void writeBoolean(boolean b)
                                    throws IOException
```

Writes the boolean **`b`** to the file

```
public final void writeUTF(String s)
                                    throws IOException
```

Writes the String **`s`** to the file using the UTF representation for Strings.

Methods in the Class **`RandomAccessFile`**

```
public final byte readByte() *
                     throws IOException
```

Reads a byte value from the file and returns that value

```
public final short readShort() *
                     throws IOException
```

Reads a short value from the file and returns that value

```
public final int readInt() *
                     throws IOException
```

Reads an int value from the file and returns that value

Methods in the Class **`RandomAccessFile`**

```
public final float readFloat() *
                        throws IOException
```

Reads a float value from the file and returns that value

```
public final double readDouble() *
                        throws IOException
```

Reads a double value from the file and returns that value

```
public final char readChar() *
                        throws IOException
```

Reads an char value from the file and returns that value

Methods in the Class **RandomAccessFile**

```
public final boolean readBoolean() *
                              throws IOException
```

Reads a boolean value from the file and returns that value

```
public final String readUTF() *
                              throws IOException
```

Reads a string from the file and returns that value

* all read... methods in this class can also throw an **`EOFException`** if reading past the end of the file.

Homework

- Finish all late labs and homework
- Complete Lab 6 – Binary and Object files
- Homework for Module 6 – projects 1 and 2

Group Lab 6

Reading and writing binary files containing objects of type `Loan`.