Lecture: File I/O (Part 2): Binary files

Writing

Binary files store data in the same format used by computer memory to store the values of variables.

Minimal 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 between "big endian" and "little endian" computers.

- 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 with the same object definitions.

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;
```

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.

After opening the file, `ObjectOutputStream` methods can be used to write to the file.

Primitive values can be written by methods such as `writeInt`, `writeDouble`, `writeChar`, and `writeBoolean`.

Note that `writeInt` will not convert an integer into a decimal form, and output a sequence of digits the way an integer would be written to a text file. Instead, it will directly write the four bytes that are

used to store the integer. However, it will output the most significant byte first, regardless of which is stored first in memory. That is how it remains portable between little-endian and big-endian computers.

Note also that writeChar writes a two-byte value. This is unlike many languages such as C/C++ in which a character is a single byte. That is because java uses UTF-16 internally as a way to represent unicode characters. You can instead output using UTF-8, by `writeUTF(String s)`. If all of your characters are ASCII characters (anything you can type with a US or Australian keyboard layout), then the UTF-8 representation is the same as the ASCII representation.

The stream should always be closed after writing.

Some methods in the class `ObjectOutputStream`

public ObjectOutputStream (OutputStream streamObject)

This is no constructor that takes a file name as an argument. If you want to cera create a stream using a file name, you use

```
        new ObjectOutputStream (new FileOutputStream(fileName))
```

This creates a blank file. If there already is a file named fileName, then the old contents of the fiel are lost.

If you want to create a stream using an object of the class File, you use

```
        new ObjectOutputStream (new FileOutputStream (fileObject))
```

The constructor for FileOutputStream may throw a FileNotFoundException, which is a kind of IOException. If the FileOutputStream constructor succeeds, then the constructor for ObjectOutputStream may throw a different IOException.

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

Writes the int value n to the output stream.

```
public void writeShort(short n) throws IOException
public void writeLong(long n) throws IOException
public void writeFloat(float x) throws IOException
public void writeDouble(double x) throws IOException
public void writeBoolean(boolean b) throws IOException
public void writeByte(int b) throws IOException
```

Writes the value of the corresponding type to the output stream

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

Writes the char value n to the output stream. Note that it expects its argument to be an int value. However, if you simply use the char value, then Java will automatically type cast it to an int value. The following are equivalent:

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

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

Output aString in its raw representation, as a sequence of 16-bit values, high byte first.

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

Writes the (almost) UTF-8 representation of aString to the output stream. UTF-8 refers to a particular method of encoding the string. Java uses a slight variant of UTF-8, described at [https://docs.oracle.com/javase/7/docs/api/java/io/DataInput.html#modified-utf-8] To read the string back from the file, you should use the method `readUTF` of the class `ObjectInputStream`.

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

Writes its argument to the output stream. The object argument should be an object of serializable class, a concept discussed in Chapter 10 of the text book.

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

Closes the stream's connection to a file. This method calls `flush` before closing the file.

```
public void flush() throws IOException
```

Flushes the output stream. This forces an actual physical write to the file of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke `flush`.

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

public class BinaryOutputDemo
{
    public static void main(String[] args)
    {
        ObjectOutputStream outputStream = null;
        String filename = "numbers.dat";

        try
        {
            outputStream = new ObjectOutputStream(new FileOutputStream(filename));

            int i;
            for(i=0; i<10; i++)
```

```
                outputStream.writeInt(i);

            System.out.println("Numbers written to: " + filename);
            outputStream.close();
        }
        catch (IOException e)
        {
            System.out.println("Could not write to file: " + filename);
            System.exit(0);
        }
    }
}
```

Exercise: Find the length of the file after each writeInt call. Replace `writeInt` by `writeChar` or `writeLong` and compare the file lengths.

Reading

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;
```

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.

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` or `readChars` is used to input values of type String, depending on whether `writeUTF` or `writeChars` was used to write the file.

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.

Some methods in the class ObjectInputStream

```
public ObjectInputStream(InputStream streamObject)
```

There is no constructor that takes a file name as an argument. If you want to create a stream using a file name, you use

```
        new ObjectInputStream(new FileInputStream(fileName))
```

Alternatively, you can use an object of the class File in place of the `fileName`, as follows:

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

The constructor for `FileInputStream` may throw a `FileNotFoundException`, which is a kind of `IOexception`. If the `FileInputStream` constructor succeeds, then the constructor for `ObjectInputStream` may throw a different `IOException`.

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

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

```
public int readShort() throws IOexception
public long readLong() throws IOexception
public double readDouble() throws IOexception
public float readFloat() throws IOexception
public char readChar() throws IOexception
public boolean readBoolean() throws IOexception
public String readUTF() throws IOexception
public String readChars() throws IOexception
```

The same description as above applies, with the appropriate types substituted.

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

Reads an object from the input stream. The object read should have been written using writeObject of the class ObjectOutputStream. Throws a ClassNotFoundException if the serialized object in the input stream doesn't match any known object type in the current program. If an attempt is made to read beyond the end of the file, and EOFException is thrown. May throw variaous other IOExceptions.

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

Skips `n` bytes. Returns the number of byes skipped. This may be less than `n` if, for example, the end of file is reached before skipping `n` bytes. This never throws `EOFException`.

```
public void close throws IOException
```

Closes the stream's connection to a file.

Here is the example of writing a binary file from the previous slide. There are two exercise below it.

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

import java.io.ObjectInputStream;
```

```
import java.io.FileInputStream;

public class BinaryOutputDemo
{
    public static void main(String[] args)
    {
        ObjectOutputStream outputStream = null;
        String filename = "numbers.dat";

        try
        {
            outputStream = new ObjectOutputStream(new FileOutputStream(filename));

            int i;
            for(i=0; i<10; i++)
                outputStream.writeInt(i);

            System.out.println("Numbers written to: " + filename);
            outputStream.close();

            ObjectInputStream inputStream
                    = new ObjectInputStream(new FileInputStream(filename));

            for (i = 0; i < 5; i++) {
                System.out.println(inputStream.readLong());
            }
            inputStream.close();
        }
        catch (IOException e)
        {
            System.out.println("Could not write to file: " + filename);
            System.exit(0);
        }
    }
}
```

Exercise: Add code after the `outputStream.close()`, that will read in the integers using the `ObjectInputStream` method `readInt` and print them out.

Exercise: Repeat using readShort. This should give the sequence 0, 0, 0, 1, 0, 2, 0, 3, 0, 4. Why?

Exercise: Repeat using readLong. You will need to reduce the "i<10" to "i < 5". (Why?) This should give output 1, 8589934595, 17179869189, 25769803783, 34359738377. Why? Try converting these numbers to hexadecimal, possibly using the web site https://www.rapidtables.com/convert/number/decimal-to-hex.html

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.

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

public class EOFDemo {

    public static void main(String[] args) {
        ObjectInputStream inputStream = null;
        String filename = "numbers.dat";

        try {
            inputStream = new ObjectInputStream(new FileInputStream(filename));

            int number;
            System.out.println("Reading numbers from file: " + filename);
            try {
                while(true) {
                    number = inputStream.readInt();
                    System.out.println(number);
                }
            }
            catch (EOFException e) {
                System.out.println("No more numbers in file.");
            }

            inputStream.close();
        } catch (FileNotFoundException e)
        {
            System.out.println("Could not open file: " + filename);
        } catch(IOException e) {
            System.out.println("Could not read from file.");
        }
    }
}
```

Exercise: Modify the above to create numbers.dat containing 10 integers, before the code tries to read it.

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();
```

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.

```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.Serializable;
import java.io.File;

public class Player implements Serializable {

    public static double DEFAULT_HITPOINTS = 100;
    public static String DEFAULT_NAME = "Anonymous";

    private String name;
    private double hitpoints;
    private boolean flightAttack;

    public Player() {
        name = DEFAULT_NAME;
        hitpoints = DEFAULT_HITPOINTS;
        flightAttack = false;
    }

    public Player(String name, double hitpoints, boolean flightAttack)     {
        if(name.equals(""))         {
            this.name = DEFAULT_NAME;
```

```
        } else
            this.name = name;

        if(hitpoints<=0) {
            System.out.println("WARNING: new character created with default hitpoints.");
            this.hitpoints = DEFAULT_HITPOINTS;
        } else
            this.hitpoints = hitpoints;

        this.flightAttack = flightAttack;

        // System.out.println("+new Player created: " + name + ", hitpoints: " + hitpoints + ", fli
    }

    public Player(Player otherPlayer) {
        if(otherPlayer==null) {
            System.out.println("ERROR: player cannot be null");
            System.exit(0);
        }

        name = otherPlayer.name;
        hitpoints = otherPlayer.hitpoints;
        flightAttack = otherPlayer.flightAttack;
    }

    // GETTERS
    public String getName()    { return name; }
    public double getHitpoints() { return hitpoints; }
    public boolean getFlightAttack() { return flightAttack; }

    // SETTERS / MODIFIERS

    public void setName(String name) {
        if(name.equals(""))
            this.name = DEFAULT_NAME;
        else
            this.name = name;
    }

    public void setHitpoints(double hitpoints) {
        if(hitpoints<=0) {
            System.out.println("WARNING: new character created with default hitpoints.");
            this.hitpoints = DEFAULT_HITPOINTS;
        }
        this.hitpoints = hitpoints;
    }

    public void setFlightAttack(boolean flightAttack) {
        this.flightAttack = flightAttack;
    }

    public String toString() {
        return name + " with current hitpoints: " + hitpoints + ", can attack flying objects: " + f
    }
```

```
    public boolean equals(Player otherPlayer) {
        if(otherPlayer == null) {
            System.out.println("ERROR: player cannot be null");
            System.exit(0);
        }
        // else
        return (name.equals(otherPlayer.name) && hitpoints==otherPlayer.hitpoints && flightAttack==
    }

    public static void main(String[] args)     {
        String filename = "player.dat";
        File fileObject = new File(filename);

        if(fileObject.exists()) {
            System.out.println(filename + " already exists. Loading player...");
            ObjectInputStream inputStream = null;

            try {
                inputStream = new ObjectInputStream(new FileInputStream(filename));
                Player myPlayer = (Player) inputStream.readObject();

                System.out.println(myPlayer.toString());
            } catch (FileNotFoundException e) {
                System.out.println("Could not open file: " + filename);
            } catch(IOException e) {
                System.out.println("Could not read from file.");
            } catch(ClassNotFoundException e) {
                System.out.println("Class not found.");
            }
        } else  {
            ObjectOutputStream outputStream = null;
            try {
                outputStream = new ObjectOutputStream(new FileOutputStream(filename));

                Player myPlayer = new Player("Ragnar", 100, false);
                outputStream.writeObject(myPlayer);

                outputStream.close();
            } catch (FileNotFoundException e) {
                System.out.println("Could not open file: " + filename);
            } catch(IOException e) {
                System.out.println("Could not read from file.");
            }
        }
    }

}
```

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.

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();
```

Reading and writing the same file

Random access: reading and writing to the same file

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.

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 kind of marker called the *file pointer* that is always positioned at one of the bytes. 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`.

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`, and `writeUTF` for output.

It does not have `writeObject` or `readObject` methods, however.

Opening a file for random access

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 read form the file, but not write to it
- "`rws`" or "`rwd`" (See Table of methods from `RandomAccessFile`)

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.

Some methods from class RandomAccessFile

```
public RandomAccessFile(String fileName, String mode)
public RandomAccessFile(File fileObject, String mode)
```

Opens the file, does not delete data already in the file, but does position the file pointer at the first (zeroth) location.

The mode must be one of the following:

- "r" Open for reading only.
- "rw" Open for reading and writing.
- "rws" Same as "rw", and also requires that every update to the file's content *or metadata* be written synchronously to the underlying storage device.
- "rwd" Same as "rw", and also requiring that every update to the file's content be written synchronously to the underlying storage device.
  ("rws" and "rwd" are not covered in this course.)

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

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

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

Moves the file pointer to the specified location.

```
public log length() throws IOException
```

Returns the length of the file, in bytes.

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

Sets the length of this file.

If the present length of the file as returned by the length method is greater than the newLength argument, then the file will e truncated. In this case, if the file pointer location as returned by the getFilePointer method is greater than newLength, then after this method returns, the file pointer location will be equal to newLength.

If the present length of the file as returned by the length method is smaller than newLength, then the file will be extended. In this case, the contents of the extended portion of the file are not defined.

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

Closes the stream's connection to a file.

```
public void write(int b) throws IOException
```

Writes the specified byte to the file.

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

Writes `a.length` bytes from the specified byte array to the file.

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

Writes the byte b to the file.

```
public final void writeShort(short n) throws IOException
public final void writeInt(int n) throws IOException
public final void writeLong(long n) throws IOException
public final void writeFloat(float f) throws IOException
public final void writeDouble(double d) throws IOException
public final void writeChar(char c) throws IOException
public final void writeBoolean(boolean b) throws IOException
```

Writes the appropriate type to the file.

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

Writes the String s to the file, using almost-UTF-8 encoding.

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

Reads a byte of data from the file and returns it as an integer in the range 0-255.

```
public int read(byte[] a) throws IOException
```

Reads `a.length` bytes of data from the file into the array of bytes a. Returns the number of bytes read or -1 if the end of the file is encountered.

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

Reads a byte value from the file and returns that value. If an attempt is made to read beyond the end of the file, and `EOFException` is thrown.

```
public final short readShort() throws IOException
public final int readInt() throws IOException
public final long readLong() throws IOException
public final float readFloat() throws IOException
public final double readDouble() throws IOException
public final char readChar() throws IOException
public final boolean readBoolean() throws IOException
```

As above, with "byte" replaced by the appropriate type.

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

Reads a String value from the file, coded with almost-UTF-8, and returns that value. If an attempt is made to read beyond the end of the file, and `EOFException` is thrown.