Chapter 19

Binary I/O

Introduction

- Inside of a computer, all data is stored in binary.
- Knowing how to read and write binary files allows us to create a file format that is proprietary to our program.
- Wring binary also allows us to use compression and other techniques not available in plain text.
- Binary files can also hold “snap shots” of running classes so they can be loaded later, or by another JVM

The File Class

The File class is a class that wraps the File Descriptor of any File that is on the file system (or is going to be).

The File class only contains a pointer to the path, and all of the info about the file, but does not allow us to read from or write to that file.

Lets look at what the File class offers us, and see how to use it.

The File Class

java .io.file

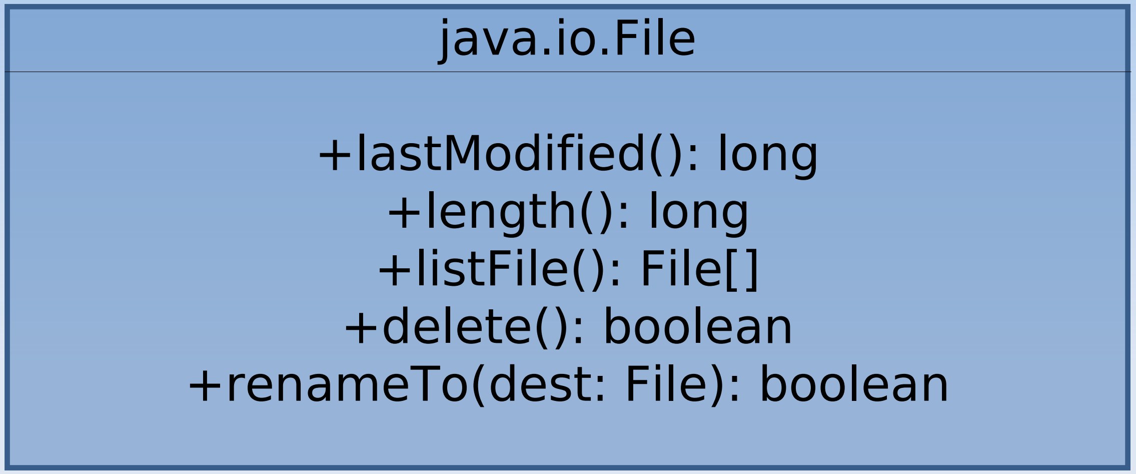
+File(pathname: String): File
+File(parent: String, child: String): File
+File(parent: File, child: String): File

+exists(): boolean
+canRead(): boolean
+canWrite(): boolean
+isDirectory(): boolean
+isFile(): boolean
+isAbsolute(): boolean
+isHidden(): boolean

+getAbsolutePath(): String
+getCanonicalPath(): String
+getName(): String
+getParent(): String

The Canonical path is simply the system independent representation of the absolute path. For example on a windows system the Canonical path begins with C:\ But on a Unix system, it would simply be a / symbol.

The File Class



```
java.io.File

+lastModified(): long
+length(): long
+listFile(): File[]
+delete(): boolean
+renameTo(dest: File): boolean
```

As you can see by the previous slide and this one, there are a ton of methods that can be used to find out all kinds of useful information about a specific file.

The File Class

```
import java.io.File;

public class TestFile{
    public static void main(String[] args){
        File file = new File("myFile.txt");
        System.out.println("Does it exist? " + file.exists());
        System.out.println("The file has " + file.length() + " bytes");
        System.out.println("Can it be read? " + file.canRead());
        System.out.println("Can you write to it? " + file.canWrite());
        System.out.println("Is it a directory? " + file.isDirectory());
        System.out.println("Is it a file? " + file.isFile());
        System.out.println("Is it absolute? " + file.isAbsolute());
        System.out.println("Is it hidden? " + file.isHidden());
        System.out.println("Absolute path: " + file.getAbsolutePath());
        System.out.println("Last modified on: " + new
                                   java.util.Date(file.lastModified()));
    }
}
```

Try out this example, and run it with a file that exists, and one that doesn't

Basic I/O

- Well thats all good and dandy, but what about if you want to write any data to the file?
- The most basic I/O operations in Java are Text I/O operations.

  (Note: I mean basic as in easy to use, and fundamental to a programmer, not basic as in primitive or lowest level)
- To do this basic I/O use use PrintWriter and Scanner.

PrintWriter

The PrintWriter can be created in two ways, The first is with just a file name string

```
PrintWriter writer = new PrintWriter(“someFile.txt”);
```

The second way is to use a File object

```
File fileToWrite = new File(“someFile.txt”);
PrintWriter writer = new PrintWriter(fileToWrite);
```

Either way we create a PrintWriter, it will automatically create and open the file specified if it does not exist. If it does exist, it will OVERWRITE it! So be careful if you don’t want overwrite behavior.

PrintWriter

```
import java.io.*;

public class TestPrintWriter{
    public static void main(String[] args) throws Exception{
        PrintWriter writer = new PrintWriter("someFile.txt");

        for (int i = 0; i < 20; i++){
            writer.print(i + " ");
        }

        writer.println();
        writer.println("Now we are writing out output to a file!");

        writer.println("There are overloaded print, println, and “ +
                                "printf methods in the printWriter");
        writer.printf("%2.4f %c, %d", 3.1415, '@', 130);

        writer.flush();    //flush the output
        writer.close();    //close the file
    }
}
```

PrintWriter

- Because the print, printf, and println methods in the PrintWriter class exactly match the print, printf, and println methods that we use to write text to the console, we can easily migrate our output from the screen to a file with little effort.
- This is the beauty of the PrintWriter class, and why I consider it to be a very basic operation for any programmer.

Basic I/O

- Okay, so now that we can write text to a file, how do we read the text from that file?
- Actually, you do it the same way you read text from the console!
- We will use a scanner, but instead of scanning System.in, We will scan a File.
- Keep in mind when scanning a file, The scanner class can throw IOExceptions which are checked exceptions. Remember to handle or declare.

Scanner for reading files

```
import java.util.Scanner;

public class TestScanner{
     public static void main(String[] args) throws Exception{
          int[] intArray = new int[100];
          String[] stringArray = new String[100];
          double[] doubleArray = new double[100];
          int i = 0;
          int j = 0;
          int k = 0;
          java.io.File file = new java.io.File("someFile.txt");
          Scanner input = new Scanner(file);

          while(input.hasNext()){
               if (input.hasNextInt()){
                    intArray[i++] = input.nextInt();
               }
               else if (input.hasNextDouble()){
                    doubleArray[j++] = input.nextDouble();
               }
               else{
                    stringArray[k++] = input.next();
               }
          }
          input.close();

          for (int m = 0; m < 100; m++){
               System.out.println(intArray[m] + " " + doubleArray[m] + " " + stringArray[m]);
          }
     }
}
```

Text I/O

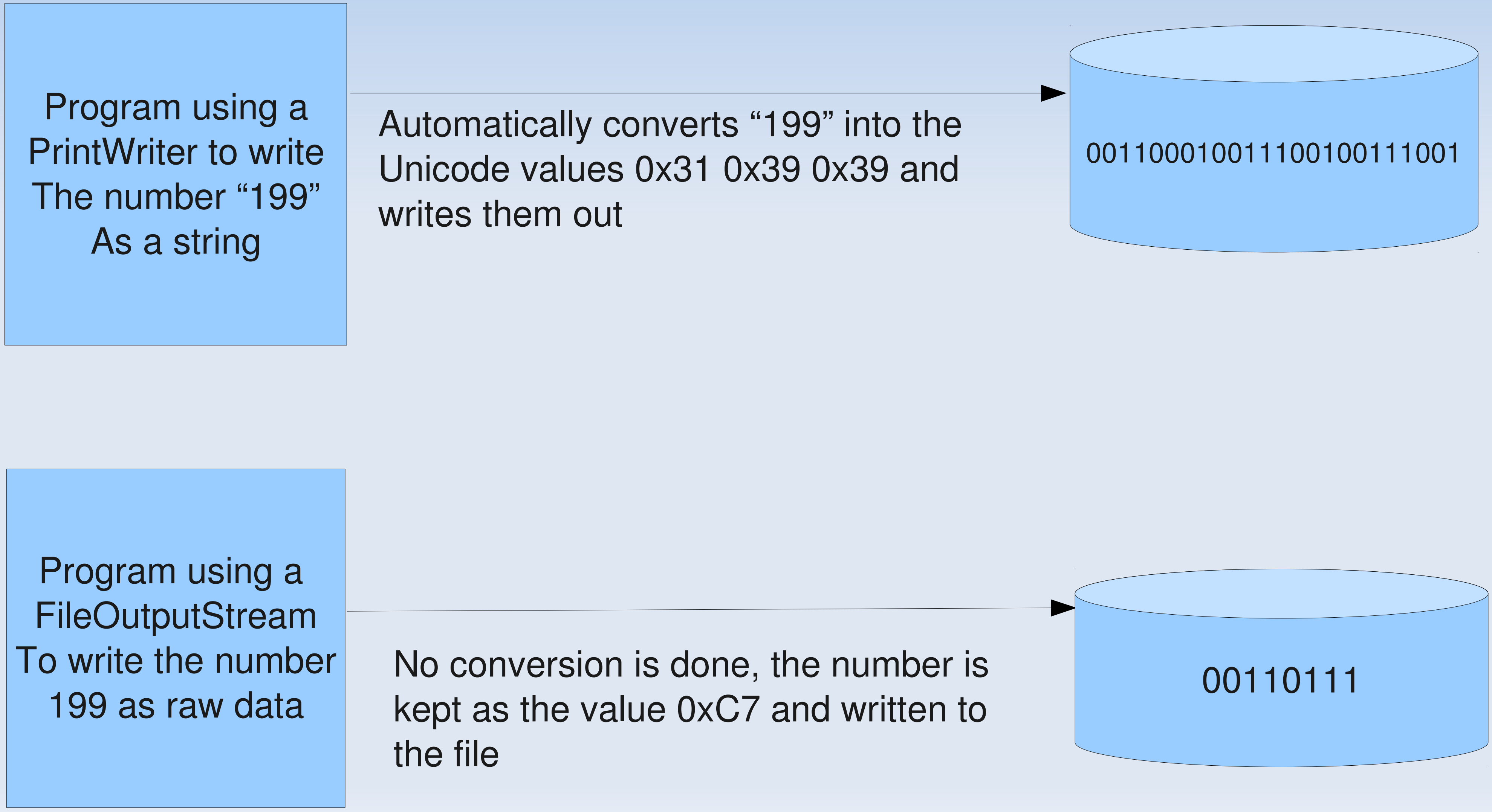
- Text I/O is the simplest, and most straight forward to understand.
- You also have the added benefit of being able to read the files once they have been written!
- The next thing we will be looking at is Binary I/O.
- Although technically all I/O on a computer is binary I/O, the text I/O classes perform a hidden layer of encoding and decoding for us.

Binary I/O

- Java treats all Binary I/O operations as streams of bits (1's and 0's)
- Like I mentioned in the previous slide, Text I/O is built on binary I/O but adds a lay of encoding and decoding that is otherwise necessary for a programmer to do.
- Lets look at an example of the difference between binary and text I/O.

Binary Vs. Text I/O

Lets write the number 199 to a File using both text and binary I/O


Program using a PrintWriter to write The number “199” As a string
Automatically converts “199” into the Unicode values 0x31 0x39 0x39 and writes them out
001100010011100100111001
Program using a FileOutputStream To write the number 199 as raw data
No conversion is done, the number is kept as the value 0xC7 and written to the file
00110111

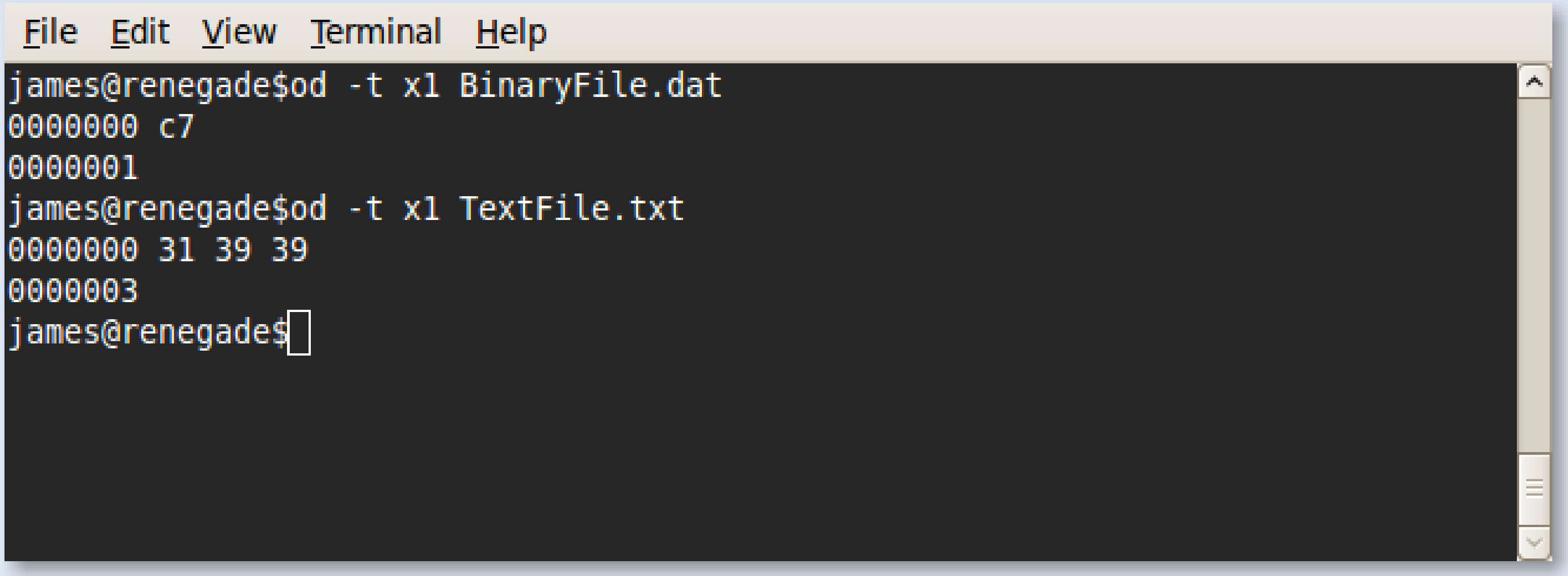
Binary Vs. Text I/O

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

public class IOTest{
    public static void main(String[] args) throws Exception{
        PrintWriter writer = new PrintWriter("TextFile.txt");
        File binaryFile = new File("BinaryFile.dat");
        FileOutputStream fOut = new FileOutputStream(binaryFile);

        int i = 199;

        writer.print(i);
        fOut.write(i);

        writer.close();
        fOut.close();
    }
}
```

This program results in the creation of two files, The first file contains the number 199 stored as a readable string. The second contains the number 199 as its raw byte value.

Binary Vs. Text I/O

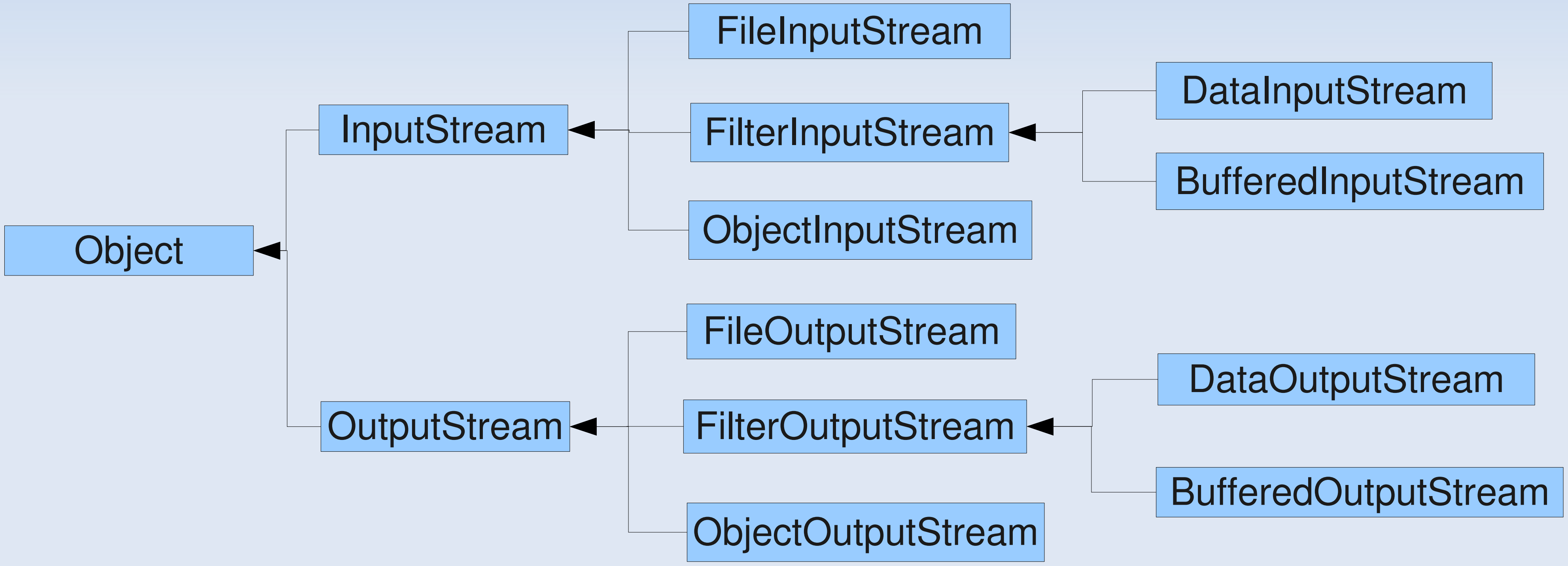
- We can use the Unix program `od` (Octal Dump) to look at the values stored inside of each file



```
james@renegade$od -t x1 BinaryFile.dat
0000000 c7
0000001
james@renegade$od -t x1 TextFile.txt
0000000 31 39 39
0000003
james@renegade$
```

Binary I/O

- Lets dive into binary I/O to see how to use it, and how to make it work for us.



Binary I/O

- The abstract classes InputStream and OutputStream defined the most fundamental I/O operations available in Java
- NOTE: all the methods in the binary I/O classes are declared to throw a java.io.IOException or a subclass

java.io.InputStream

read(): int
read(b: byte[]): int
read(b: byte[], offset; int, length: int): int

avilable(): int
close(): void
skip(n: long): long

markSupported(): boolean
mark(readlimit: int): void
reset(): void

java.io.OutputStream

write(b: int): void
write(b: byte[]): void
write(b: byte[], offset: int, length: int): void

close(): void
flush(): void

FileInputStream / FileOutputStream

- The FileInputStream and FileOutputStream are simply implementations of the InputStream and OutputStream classes.
- They add NO new methods.

java.io.FileInputStream

FileInputStream(file: File)
FileInputStream(filename: String)

java.io.FileOutputStream

FileOutputStream(file: File)
FileOutputStream(filename: String)
FileOutputStream(file: File, append: boolean)
FileOutputStream(filename: String, append: boolean)

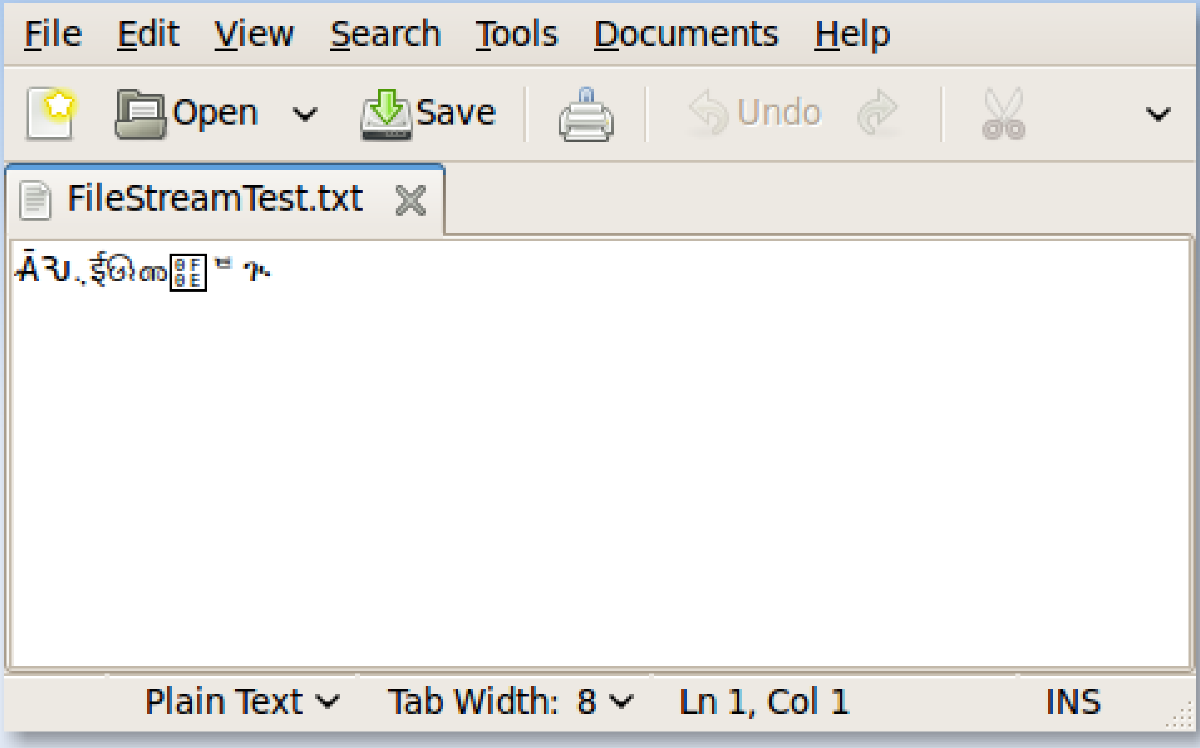
FileInputStream / FileOutputStream

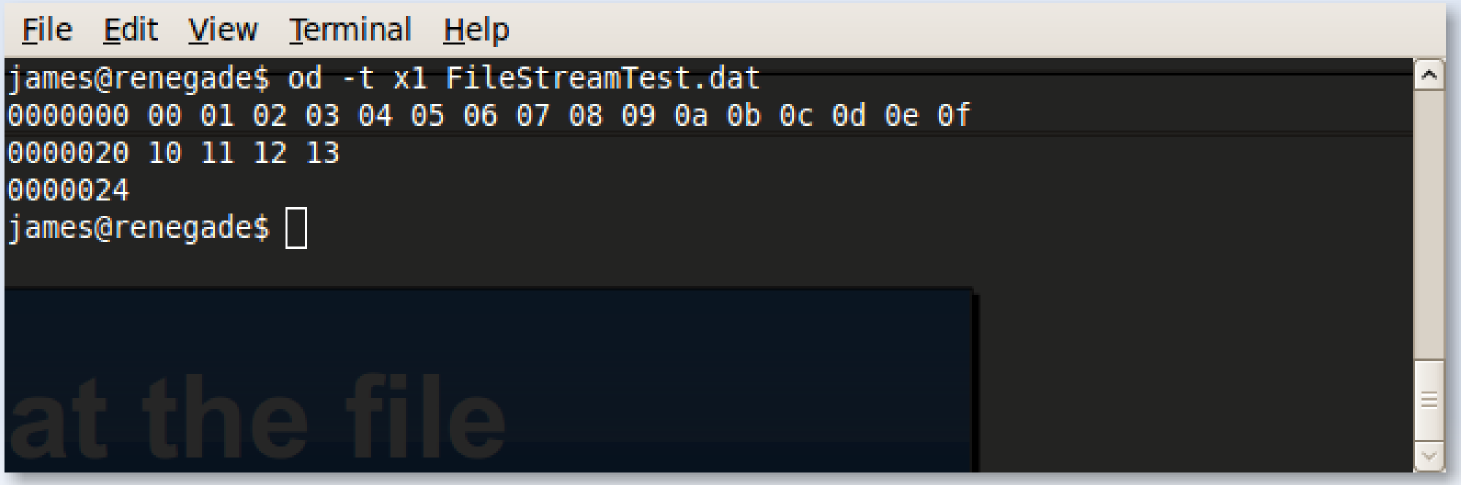
- If you attempt to read a file that does not exist, The input stream will throw a FileNotFoundException
- If you write to a file that doesn't exist, it will create it for you.
- When writing to a file; If the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason then a FileNotFoundException is thrown.

FileInputStream / FileOutputStream

```
import java.io.*;

public class FileStreamTest{
    public static void main(String[] args) throws Exception{
        File file = new File("FileStreamTest.dat");
        FileOutputStream fout = new FileOutputStream(file);

        for (int i = 0; i < 20; i++){
            fout.write(i);
        }

        fout.close();

        FileInputStream fin = new FileInputStream(file);

        int nextUnsignedByte = 0;

        while((nextUnsignedByte = fin.read()) != -1){
            System.out.println("Read " + nextUnsignedByte);
        }

        fin.close();
    }
}
```

Closer look at the file





```
james@renegade$ od -t x1 FileStreamTest.dat
0000000 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0000020 10 11 12 13
0000024
james@renegade$
```

Tips

- You can use a FileInputStream as an argument to a scanner constructor. This allows you to scan a binary file for more specific data.
- You can also use a FileOutputStream as an argument to a PrintWriter. This will allow you to append text to the end of another file.
- Simply use

```
new PrintWriter(new FileOutputStream(“file.txt”, true));
```

DataInputStream / DataOutputStream

- The DataInputStream and DataOutputStream are FilteredStreams. This means that they contain extra methods that filter the byte of data being read in from the file to find specific items.

java.io.DataInputStream
readBoolean(): boolean
readByte(): byte
readChar(): char
readFloat(): float
readDouble(): double
readInt(): int
readLong(): long
readShort(): short
readLine(): String
readUTF(): String

java.io.DataOutputStream
writeBoolean(b: boolean): void
writeByte(v: int): void
writeBytes(s: String): void
writeChar(c: char): void
writeChars(s: String): void
writeFloat(v: float): void
writeDouble(v: double): void
writeInt(v: int): void
writeLong(v: long): void
writeShort(v: short): void
writeUTF(s: String): void

DataInputStream / DataOutputStream

```
import java.io.*;

public class TestDataStreams{
    public static void main(String[] args){
        File file = new File("DataStream.dat");
        DataOutputStream out = null;
        DataInputStream in = null;

        try{
            out = new DataOutputStream(new FileOutputStream(file));

            String[] students = {"James", "John", "Mary", "Edward", "Mike"};
            int[] scores = {98, 100, 95, 72, 83};

            for (int i = 0; i < students.length && i < scores.length; i++){
                out.writeUTF(students[i]);
                out.writeInt(scores[i]);
            }

        }catch (IOException ioe){
            ioe.printStackTrace();
        }finally{
            try{
                out.close();
            }catch(IOException ioe){
                ioe.printStackTrace();
            }
        }
```

DataInputStream / DataOutputStream

```
        try{
            in = new DataInputStream(new FileInputStream(file));

            while(true){
                System.out.println(in.readUTF() + " -> " + in.readInt());
            }
        }catch (EOFException eof){
            System.out.println("End of file reached");
        }catch (IOException ioe){
            ioe.printStackTrace();
        }finally{
            try{
                in.close();
            }catch(IOException ioe){
                ioe.printStackTrace();
            }
        }
    }
}
```

Strings and DataStreams

- Strings of text can be tricky with the Data Streams.
- The readLine() method is Deprecated because it doesn't work correctly, however the writeChars() method works fine so it is still okay to use.
- Lets look at the differences between the different methods to write and read strings in a binary file

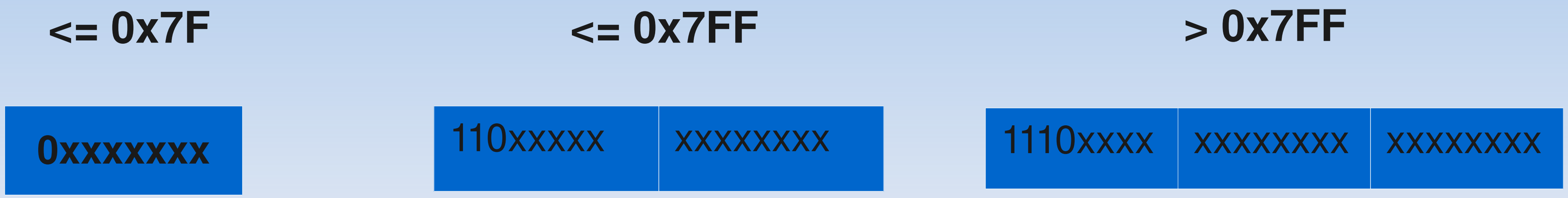
Strings and DataStreams

- writeBytes(s: String) – This method only writes the lower order byte of the Unicode for each character. This in effect converts Unicode to ASCII but can have undesirable effects when trying to write non ASCII characters to the file because it discards the high order byte.
- writeChars(s: String) – This method write the string as a sequence of 16 bit unicode characters. You can still read from this string once character at a time, as well as grab this data with a Scanner, However the readLine() method that is supplied with the DataInputStream is broken and should be avoided.

Strings and DataStreams

- writeUTF(s: String) – This method writes out the string in the UTF-8 representation for each character.
- UTF-8 allows for systems to operate with both Unicode and ASCII. Depending of the code of the character, it can store that character using 1, 2, or 3 bytes. With the beginning bits of the byte marking the length.
- UTF-8 Also stores the total number of characters contained in the string in 2 bytes preceding the string.

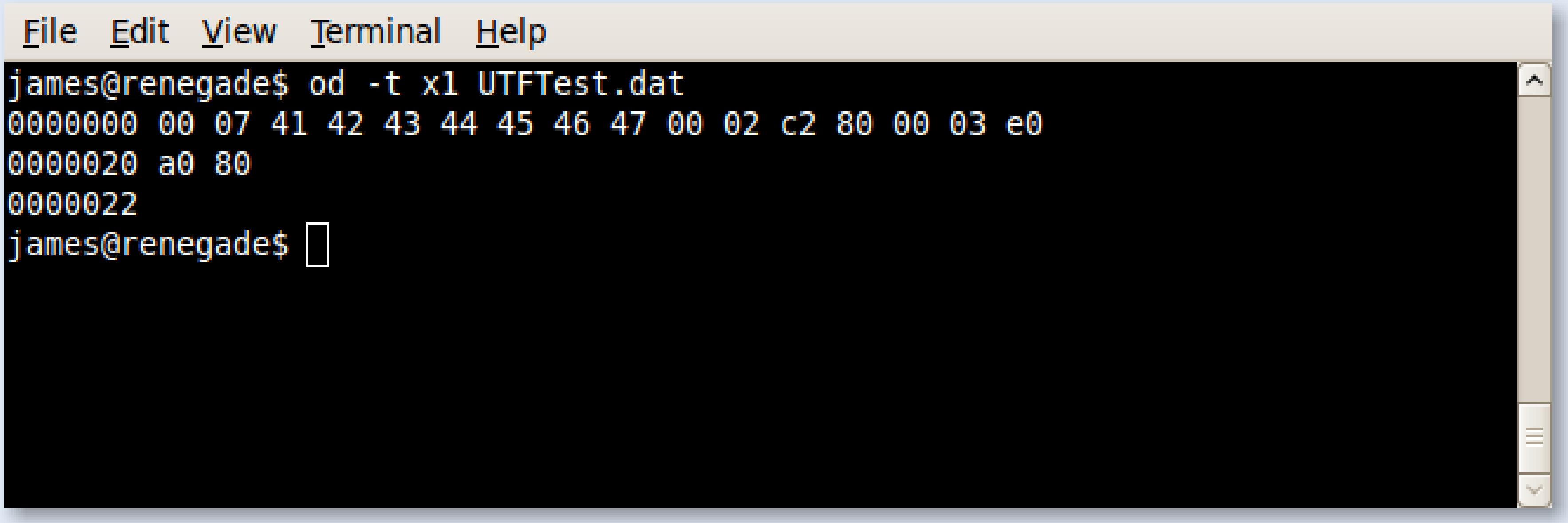
Strings and DataStreams



Stored in 1 byte with leading 0

Stored in 2 bytes with leading 110

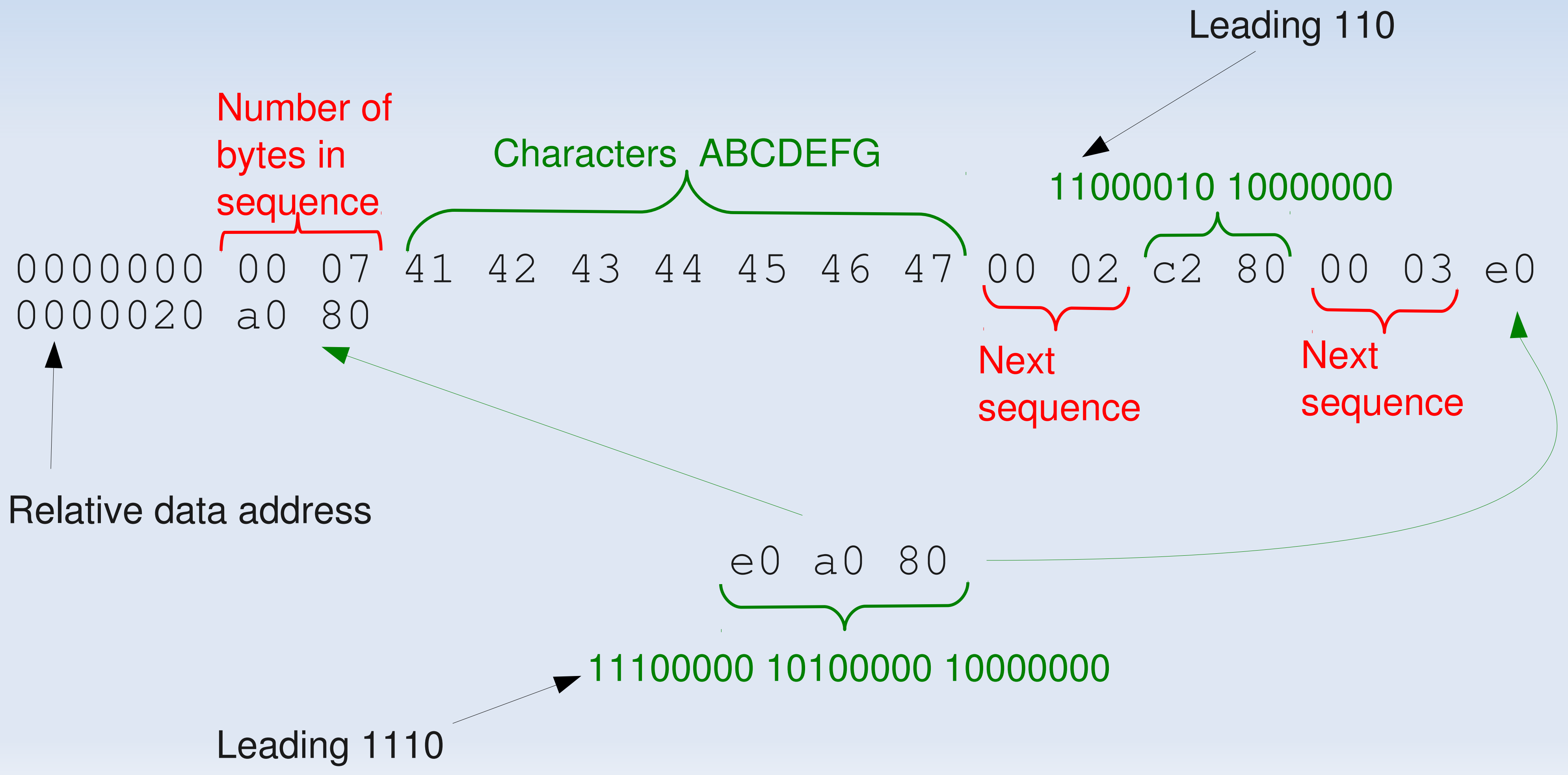
Stored in 3 bytes with leading 1110



```
james@renegade$ od -t x1 UTFTest.dat
0000000 00 07 41 42 43 44 45 46 47 00 02 c2 80 00 03 e0
0000020 a0 80
0000022
james@renegade$
```

Strings and DataStreams

- Lets break down that output



```
0000000  00  07  41  42  43  44  45  46  47  00  02  c2  80  00  03  e0
0000020  a0  80
```

Strings and DataStreams

- Using the UTF-8 methods will save you space and time if you are writing purely ASCII statements.
- If you still want to use the writeChars() method you can read those chars using a different class like so

```
FileInputStream in = new FileInputStream(someFile);
BufferedReader buff = new BufferedReader(new InputStreamReader(in));

buff.readLine();
```

BufferedInputStream / BufferedOutputStream

- BufferedInputStream, and BufferedOutputStream do not provide any new methods, They simply add a buffer to the InputStream to speed up I/O.
- You can wrap a FileInputStream with a BufferedInputStream, and a FileOutputStream with a BufferedOutputStream.
- For more control, you would normally then pass this BufferedStream to a DataStream constructor

```
DataOutputStream out = new DataOutputStream(
                            new BufferedOutputStream(
                            new FileOutputStream("UTFTest.dat")));
```

BufferedInputStream / BufferedOutputStream

- The default buffer size for a BufferedStream is 512 Bytes (½ Kb)
- A BufferedInputStream will read as much data as possible to have it ready on the buffer. This speeds up reads for large linear files.
- A BufferedOutputStream won't actually write any data until its buffer is full or the programmer calls the flush() method.

BufferedInputStream / BufferedOutputStream

- Since BufferedStreams dont add any new methods, we are not going to do an explicit example, but you should use BufferedStreams when ever you do binary I/O in order to speed up the process.
- Just wrap a FileXStream with a BufferedXStream and you are ready to go!

ObjectInputStream / ObjectOutputStream

- DataInputStream and DataOutputStream allow you to write primitive data directly to a file as binary data.
- ObjectInputStream and ObjectOutputStream allow you to write whole Objects to a file so that there current state can be saved and re-loaded later.
- The ability to do this is what makes Applets, RMI, and a host of other Java features possible!

ObjectInputStream / ObjectOutputStream

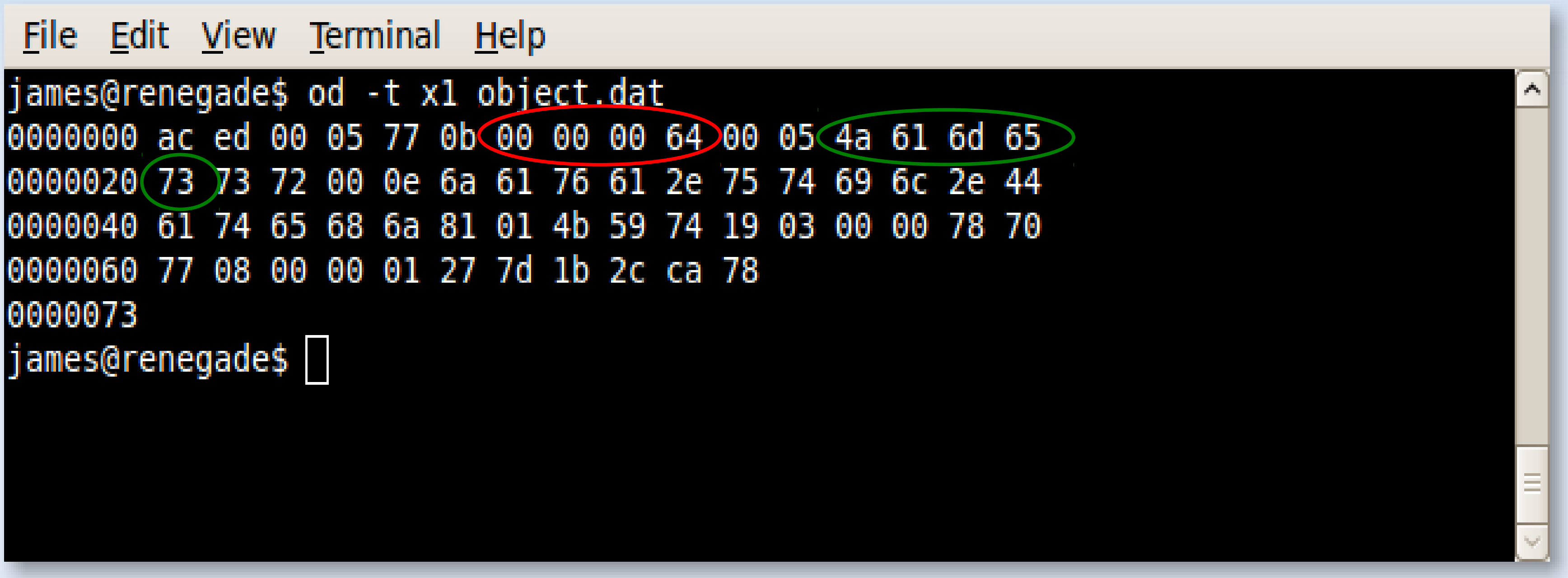
- ObjectInputStream has a readObject(): Object method
- ObjectOutputStream has a writeObject(o: Object): void method.
- You can wrap an ObjectOutputStream around a FileOutputStream, or BufferedOutputStream and save objects directly to a file. Lets take a look.

ObjectInputStream / ObjectOutputStream

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

public class SaveObject{
    public static void main(String[] args) throws Exception{
        ObjectOutputStream out = new ObjectOutputStream(
            new BufferedOutputStream(
                new FileOutputStream("object.dat")));

        int i = 100;
        String name = "James";
        Date date = new Date();

        out.writeInt(i);
        out.writeUTF(name);
        out.writeObject(date);

        out.flush(); //optional here
        out.close();
    }
}
```

ObjectInputStream / ObjectOutputStream

- Lets take a quick peek at the object.dat file



```
james@renegade$ od -t x1 object.dat
0000000 ac ed 00 05 77 0b 00 00 00 64 00 05 4a 61 6d 65
0000020 73 73 72 00 0e 6a 61 76 61 2e 75 74 69 6c 2e 44
0000040 61 74 65 68 6a 81 01 4b 59 74 19 03 00 00 78 70
0000060 77 08 00 00 01 27 7d 1b 2c ca 78
0000073
james@renegade$
```

The 4 byte int = 100

My name in UTF-8

All the rest is object data laid out by the ObjectOutputStream class.

ObjectInputStream / ObjectOutputStream

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

public class LoadObject{
    public static void main(String[] args) throws Exception{
        ObjectInputStream in = new ObjectInputStream(
            new BufferedInputStream(
                new FileInputStream("object.dat")));

        int someNumber = in.readInt();
        String someString = in.readUTF();

        Date someDate = (Date)in.readObject();

        System.out.println(someNumber + " " + someString + " " +
                                            someDate.toString());

    }
}
```

```
100 James Sat Mar 20 15:44:45 EDT 2010
```

The Serializable Interface

- The ObjectInputStream and ObjectOutputStream are not majic, If you want to serialize an object from a class that you have written then you need to do one thing.
- Implement java.io.Serializable
- Once you have done this you can now save your object using ObjectOutputStream.
- There are of course some rules and we will be discussing them as we go through an example.

The Serializable Interface

```
import java.io.*;

public class Student implements Serializable{
     private String lastName;
     private String firstName;
     private int satScore;

     public Student(){
          this("Doe", "John", -1);
     }

     public Student(String last, String first, int score){
          this.lastName = last;
          this.firstName = first;
          this.satScore = score;
     }

     public int getSatScore(){
          return this.satScore;
     }

     public String getLastName(){
          return this.lastName;
     }

     public String getFirstName(){
          return this.firstName;
     }

     public String toString(){
          return lastName + "," + firstName + " got a " + satScore;
     }
}
```

The Serializable Interface

```java
import java.io.*;

public class StoreStudent{
    public static void main(String[] args){
        Student[] students = new Student[5];
        ObjectOutputStream out = null;
        ObjectInputStream in = null;

        students[0] = new Student("Wong", "Larry", 1600);
        students[1] = new Student("Black", "Joe", 1440);
        students[2] = new Student("Weis", "David", 1600);
        students[3] = new Student("Bird", "Larry", 910);
        students[4] = new Student("Hill", "Jessica", 1320);

        try{
            out = new ObjectOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream("students.dat")));

            for (int i = 0; i < students.length; i++){
                out.writeObject(students[i]);
            }

            out.close();
        }catch(IOException ioe){
            ioe.printStackTrace();
        }

        students = null; //clear the students array
```

The Serializable Interface

```
students = new Student[5];

        try{
            File f = new File("students.dat");
            in = new ObjectInputStream(
                new BufferedInputStream(
                    new FileInputStream(f)));

            System.out.println(f.length());

            for (int i = 0; i < students.length; i++){
                students[i] = (Student)(in.readObject());
            }

            in.close();

            for (int i = 0; i < students.length; i++){
                System.out.println(students[i].toString());
            }
        }catch(ClassNotFoundException cnf){
            cnf.printStackTrace();
        }catch(EOFException eof){
            System.out.println("No more students to read");
        }catch(IOException ioe){
            ioe.printStackTrace();
        }
    }

}
```

The Serializable Interface

- Run this program and see what the result is.
- Now go back to the student class and remove the `implements Seralizable` statement, recompile and see what happens.
- Put that statement back, recompile, and go to the StoreStudent class and make the following edit.

```
students[0] = new Student("Wong", "Larry", 1600);
students[1] = new Student("Black", "Joe", 1440);
//students[2] = new Student("Weis", "David", 1600);
students[2] = students[1];
students[3] = new Student("Bird", "Larry", 910);
students[4] = new Student("Hill", "Jessica", 1320);
```

- How big is the file now? Why?

The transient keyword

- There are times when we may be using a class that is not Seralizable inside of another class that is.
- When doing this you must mark the non seralizable object as transient or you will get a java.io.NotSerializableException.
- Note that any class marked transient will not be serialized, but instead will be re-initialized into a default state.

The transient keyword

```
import java.io.*;

public class Student implements Serializable{
    private String lastName;
    private String firstName;
    private transient Grade satScore;

    public Student(){
        this("Doe", "John", -1);
    }

    public Student(String last, String first, int score){
        this.lastName = last;
        this.firstName = first;
        this.satScore = new Grade(score);
    }

    public int getSatScore(){
        return this.satScore.getScore();
    }

    public String getLastName(){
        return this.lastName;
    }

    public String getFirstName(){
        return this.firstName;
    }

    public String toString(){
        return lastName + "," + firstName + " got a " + satScore;
    }
}
```

The transient keyword

```
class Grade{
   private int score;

   public Grade(){
      this(-1);
   }

   public Grade(int score){
      this.score = score;
   }

   public int getScore(){
      return this.score;
   }
}
```

Output:

```
158
Wong,Larry got a null
Black,Joe got a null
Black,Joe got a null
Bird,Larry got a null
Hill,Jessica got a null
```

RandomAccessFile

- All of the methods of using files so far are read or write only.
- RandomAccessFiles allow us to read and write to the same file with the same object.
- RandomAccessFile has all of the same methods as the DataXStreams as well as a few new ones.
- RandomAccessFiles only allow low level control over a file and its contents, so it can be a bit more difficult to use.

RandomAccessFile

Remember that RandomAccessFiles and both read and write to the file that is open. You can think of RandomAccessFiles like old Reel-To-Reel tapes that read back and fourth.

java.io.RandomAccessFile
RandomAccessFile(file: File, mode: String) RandomAccessFile(name: String, mode: String) close(): void getFilePointer(): long length(): long seek(pos: long): void setLength(newLength: long): void skipBytes(n: int): int write() - Same writes as DataOutputStream read() - Same reads as DataInputStream

RandomAccessFile

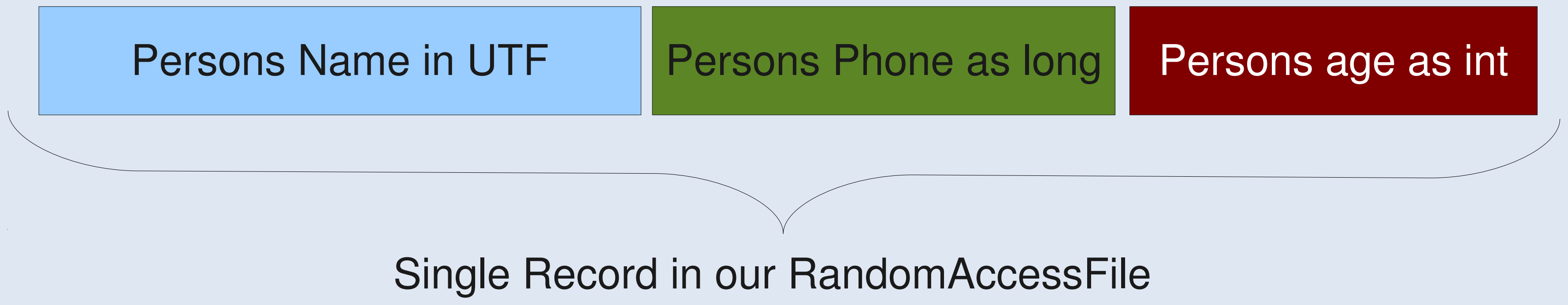
```
import java.io.*;

public class RandomAccessTest{
    public static void main(String[] args) throws Exception{
        //use mode rw for read + write, use mode r for just read
        RandomAccessFile inout = new RandomAccessFile("randomAccess.dat", "rw");

        for (int i = 0; i < 200; i++){
            inout.writeInt(i);
        }
        System.out.println("The file length is " + inout.length() + " bytes");
        inout.seek(0);

        for (int i = 0; i < 100; i++){
            int temp = inout.readInt();
            temp *=2;
            inout.seek(inout.getFilePointer() - 4); //go back to overwrite location
            inout.writeInt(temp);
            inout.skipBytes(4);  //skip every other number
        }

        inout.seek(inout.length());
        inout.writeInt(9999);

        System.out.println("The file length is now " + inout.length() + " bytes");
        inout.seek(0);
        long bytesRead = 0;

        while (bytesRead < inout.length()){
            System.out.println(inout.readInt());
            bytesRead += 4;
        }
    }
}
```

RandomAccessFile

- As you can see even in this simple example, RandomAccessFiles are incredibly useful for certain live write, and live seek situations.
- A real world example of this is MP3 files. MP3 files use predefined blocks of 4 bytes to define the header, followed by a variable number of bits for 1 second of music data. Bits 17 through 20 define how many bits are used after the header for music (between 64 – 320 kbits standard).
- So you can seek (skip) through an MP3 by reading the 32 bit header, reading the bit rate, and skipping that many bytes until the next header. This will allow 1 second skips through the music.

RandomAccessFile

- Lets try our own example (Not with MP3's)
- We will make the following specification for an address book



- Refer to handout for implementation

Lab Assignment

- Page 673 # 19.11** Splitting files
- Page 673 # 19.12** Combining files

Homework

- Page 673 # 19.20** Binary editor

Acknowledgments

Introduction to Java Programming by Y. Daniel Liang