Object Oriented Programming

Week 9 Part 2
Types of Streams

Lecture

- More on Streams
- Byte Streams
- Character Streams
- Data Streams
- Object Streams

More on Streams

Type of Streams

- Determines how the bits are interpreted
  - Byte Streams: sequence of eight bit bytes
    - Most primitive type
  - Character Streams: sequence of Unicode characters
    - May be ASCII, but allows other type of script
    - Interpretation of characters depends on localization
  - Data Streams: sequence of primitive values
    - boolean, byte, short, int, long, float, double
    - String is the only type of object that can be written
  - Object Streams: sequence of object
    - Objects with Serializable interface
      - reference are complicated

Byte Streams

- InputStream: byte stream input
  - “public abstract class InputStream extents Object implements Closeable”
  - Subclasses of InputStream must define a method that returns the next byte
- OutputStream: byte stream output
  - “public class OutputStream extends Object implements Closeable, Flushable”
  - Subclass of OutputStream must define a method that writes out a byte of data

OutputStream methods

- “PrintStream format (String f, Object … args)”
  - Like the C print command
  - Returns the PrintStream that called it
- “void print(x)”
  - Writes character representation of whatever is passed as x; calls toString, if X is an object
- “void println(x)”
  - Like print, but adds a newline
- “void write(int b)”
  - Writes a single byte to the stream.
- “void close()”
  - Closes the stream
- “void flush()”
  - Flushes the stream

Byte InputStream Methods

- “abstract int read()”
  - Reads the next byte in the stream
- “int read(byte[] b)”
  - Reads enough bytes to fill the array.
- “int read(byte() b, int offset, int length)”
  - Reads length number of bytes into b starting at offset.
- “void close()”
  - Closes the stream

Aside: Why not use tests?

- In the last lecture we developed tests as example to build up a specification of a Java class
  - Why not this time?
- By using standard input, we can demonstrate problems with it.
- To run using standard input, we need to Run as Java Application and provide a main

Example: Standard I/O

Method read() needs byte[]

Strings are easier to work with]

Print prompt

Read input

Convert input to string

Print “Hello Nat” on standard output

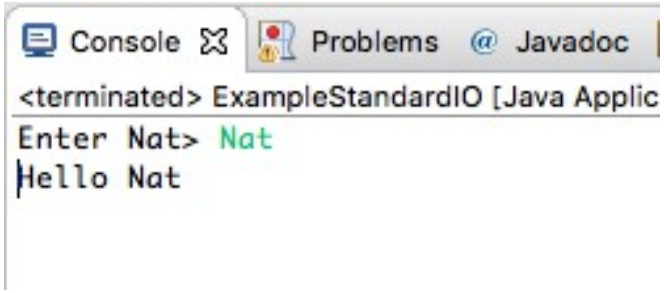
Print unexpected input on std error

Method read() throws IO exception

```
public static void main(String[] args) {
    byte[] buf = new byte[3];
    String name = "xxx";

    do {
        try {
            System.out.print("Enter Nat> ");
            System.in.read(buf, 0, 3);
            name = new String(buf);
            if (name.equals("Nat")) {
                System.out.println("Hello " + name);
            } else {
                System.err.println("Got " + name);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    } while (!name.equals("Nat"));
}
```

Example Output1



It seems to work.
Let's try something other than “Nat”

Example Output2

Enter “xxx<newline>”

Reads “xxx”; prints “Got xxx”; <newline> left

Reads <newline> into buf; now “<newline>xx

Enter “Nat”

Program quits

```
Console   Problems   @ Ja
<terminated> ExampleStandardIO [Jav
Enter Nat> xxx
Got xxxEnter Nat> Enter Nat>
Got
xx
Nat
Hello Nat
```

Standard Input is green
Standard Output is black
Standard Error is red

What happened

- The read() method only reads exactly the number of characters you tell it to.
  - It is a very primitive function, it does not help you at all with the input
- We do not want to read from byte streams

Character Streams

- This is why we need to use character streams when reading input
- Character strings also let you use non-latin alphabets
- However, character strings still only let you read one character at a time.
- A character stream will

Character InputStream Methods

- “int read()”
  - Reads the next byte in the stream
- “int read(char[] b, int offset, int length)”
  - Reads length number of chars into b starting at offset.
- “void close()”
  - Closes the stream

Character Stream Problem

- The character stream does not solve the problem we saw with byte streams.
  - We still only read character one at a time.
  - We still need to deal with the newlines as characters rather than as line terminators
- Also, the program waits for each charater
  - It does not allow type ahead
- We need to read a characters into memory before we can check for lines or tokens

Example

- StringReader(“test”);
  - Creates a character stream

```java
@Test
public void testReadWolf() {
    BufferedReader in = new BufferedReader(new StringReader("test"));
    ByteArrayOutputStream outString = new ByteArrayOutputStream();
    PrintStream out = new PrintStream(outString);
    ByteArrayOutputStream errString = new ByteArrayOutputStream();
    PrintStream err = new PrintStream(errString);

    AnimalsInOut aio = new AnimalsInOut(in, out, err);
    Wolf w = new Wolf("Meat");

    try {
        w = w.read(aio);
    } catch (IOException e) {
        fail("read() failed for Wolf");
        e.printStackTrace();
    }
    assertEquals("Wolf howls, eats test", w.toString());
    System.out.println("readWolf returned: " + w);
}
```

Buffered Streams

- Buffered streams create an area in memory into which it reads an array of characters or bytes
  - The buffer is filled as characters are available
  - The methods on a buffered stream return characters from the buffer
- Allow reading lines and tokens
- More efficient
  - Characters are read when they are available
  - The program need not wait for them

Example

- BufferedReader
  - Creates the buffer and the methods that work on the buffer
- Created from a character stream that provides the individual characters
  - StringReader(“test”)

```java
@Test
public void testReadWolf() {
    BufferedReader in = new BufferedReader(new StringReader("test"));
    ByteArrayOutputStream outString = new ByteArrayOutputStream();
    PrintStream out = new PrintStream(outString);
    ByteArrayOutputStream errString = new ByteArrayOutputStream();
    PrintStream err = new PrintStream(errString);

    AnimalsInOut aio = new AnimalsInOut(in, out, err);
    Wolf w = new Wolf("Meat");

    try {
        w = w.read(aio);
    } catch (IOException e) {
        fail("read() failed for Wolf");
        e.printStackTrace();
    }
    assertEquals("Wolf howls, eats test", w.toString());
    System.out.println("readWolf returned: " + w);
}
```

Scanners

- Scanners are object that can be created from a buffered input stream just as buffered input streams can be created from input streams
- Scanners break an input stream into tokens
  - A token is a sequence of characters that is separated by white space (i.e. space, tab and newline)
- In the following example, we can go back to using unit tests to demonstrate because we are not demonstrating input and output from the console

Example: Scanner

Create a Scanner

Read from a StringReader

Print each of the tokens twice

Tokens are separated by newlines

```
@Test
public void testScanner() {
    Scanner s = null;
    String temp = null;

    ByteArrayOutputStream outString = new ByteArrayOutputStream();
    PrintStream out = new PrintStream(outString);
    StringReader sr = new StringReader("Here are\tfour\ntokens");
    s = new Scanner(new BufferedReader(sr));

    while (s.hasNext()) {
        temp = s.next();
        out.println(temp);
        System.out.println(temp);
    }

    try {
        assertEquals(sr.read(), -1);
    } catch (IOException e) {
        e.printStackTrace();
        fail("In testScanner(), read() threw and IOException");
    }
    assertEquals("Here\nare\nfour\ntokens\n", outString.toString());
    s.close();
}
```

Output

```
Console
<terminated> Sca
Here
are
four
tokens
```

Discussion

- Input is separated by space, then tab, then newline
  - “Here are\tfour\ntokens”
- The temp string is needed the scanner removes characters from the stream when it does next()
- read() returns -1 when there is no more input
- Each token is followed by a newline because we used println(temp)
  - “Here\nare\nfour\ntokens\n”

PrintStream is a BufferedOutputStream

- Like input, output is also buffered.
- This allows the output to be formatted into a buffer before it is printed
- However, if the program crashes, everything in the buffer is lot.
- The method “flush()” tells a buffered buffered stream to release its output to the output stream
  - Important when testing crashing programs

Formatting

- Buffered output streams allow more formatting
  - PrintStream is a buffered output stream
- Buffered output streams give you
  - print()
  - println()
  - format(): a method like the C print function

Data Streams

- Data streams are not character streams
  - They return bits rather than characters or bytes
  - It is up to the program to interpret the bits
  - The bits are specified by the write and read methods
    - int readInt()
    - double readDouble()
    - String readUTF()

Data Stream Example

Set up Variables

Write out Data

Read in Data

```java
@Test
public void testDataStream() {
    DataOutputStream dos = new DataOutputStream(new BufferedOutputStream(fOut));
    DataInputStream dis = new DataInputStream(new BufferedInputStream(fIn));
    double[] ddata = {1.0, 2.1, 3.2};
    int[] idata = {1, 2, 3};
    String[] sdata = {"one", "two", "three"};
    double dtemp = 0;
    int itemp = 0;
    String stemp = "";

    try {
        for (int i = 0; i < 3; i++) {
            dos.writeDouble(ddata[i]);
            dos.writeInt(idata[i]);
            dos.writeUTF(sdata[i]);
        }
        dos.close();
    } catch (IOException e) {
        e.printStackTrace();
        fail("In testDataStream, write threw an exception");
    }

    try {
        while(true) {
            dtemp = dis.readDouble();
            itemp = dis.readInt();
            stemp = dis.readUTF();
            System.out.format("Read: %f, %d, %s%n", dtemp, itemp, stemp);
        }
    } catch (EOFException e) {
        System.out.println("Reached End of File");
    } catch (IOException e) {
        e.printStackTrace();
        fail("In testDataStream, close threw an exception");
    }
    try {
        dis.close();
    } catch (IOException e) {
        e.printStackTrace();
        fail("In testDataStream, close threw an exception");
    }
}
```

DS Example: Set Up Variables

DataOutputStream

DataInputStream

Data to be written out

Temporary variables to read data into

```
DataOutputStream dos = new DataOutputStream(
        new BufferedOutputStream(fOut));
DataInputStream dis = new DataInputStream(
        new BufferedInputStream(fIn));
double[] ddata = {1.0, 2.1, 3.2};
int[] idata = {1, 2, 3};
String[] sdata = {"one", "two", "three"};
double dtemp = 0;
int itemp = 0;
String stemp = "";
```

- DataOutputStream takes a BufferedOutputStream
  - BufferedOutputStream takes a FileOutputStream
- DataInputStream takes a BufferedInputStream
  - BufferedInputStream takes a FileInputStream

- FileOutputStream and FileInputStream are created in @Before method
- File associated with FileOutputStream and FileInputStream is deleted in @after method

DS Example: @Before and @After

@Before test

New FileOutputStream and FileInputStream

```
@Before
public void setUp() throws Exception {
    fOut = new FileOutputStream("/tmp/test.dat");
    fIn = new FileInputStream("/tmp/test.dat");
}
```

@After test

Point File to file created in @Before

Close Streams

Delete File

```
@After
public void tearDown() throws Exception {
    File f = new File("/tmp/test.dat");

    fOut.close();
    fIn.close();
    f.delete();
}
```

DS Example: Write out Data

Loop through three item in each array

Write correct type for each array

Close the stream after writing

Fail if there is an exception

```
try {
    for (int i = 0; i < 3; i++) {
        dos.writeDouble(ddata[i]);
        dos.writeInt(idata[i]);
        dos.writeUTF(sdata[i]);
    }
    dos.close();
} catch (IOException e) {
    e.printStackTrace();
    fail("In testDataStream, write threw an exception");
}
```

DS Example: Read Data Back In

Continue reading until EOF thrown

Read data into temporary variables

Print out floating point, decimal, and string

Catch EOF exception and print

Catch other exceptions and fail

Close Stream

Catch IOException and fail

```
try {
    while(true) {
        dtemp = dis.readDouble();
        itemp = dis.readInt();
        stemp = dis.readUTF();
        System.out.format("Read: %f, %d, %s%n",
                dtemp, itemp, stemp);
    }
} catch (EOFException e) {
    System.out.println("Reached End of File");
} catch (IOException e) {
    e.printStackTrace();
    fail("In testDataStream,"
            + " read threw an exception");
}
try {
    dis.close();
} catch (IOException e) {
    e.printStackTrace();
    fail("In testDataStream,"
            + " close threw an exception");
}
```

Running Data Stream Example

Output data: double, int, string

Output note: reached EOF

```
try {
    while(true) {
        dtemp = dis.readDouble();
        itemp = dis.readInt();
        stemp = dis.readUTF();
        System.out.format("Read: %f, %d, %s%n",
                dtemp, itemp, stemp);
    }
} catch (EOFException e) {
    System.out.println("Reached End of File");
} catch (IOException e) {
    e.printStackTrace();
    fail("In testDataStream,"
            + " read threw an exception");
}
try {
    dis.close();
} catch (IOException e) {
    e.printStackTrace();
    fail("In testDataStream,"
            + " close threw an exception");
}
```

Console | Problems

```
<terminated> DataStreamTest [JUr
Read: 1.000000, 1, one
Read: 2.100000, 2, two
Read: 3.200000, 3, three
Reached End of File
```

Object Streams

- It is possible to read and write objects
  - Object that will be read and write must implement the *Serializable* interface
  - Objects input from ObjectInputStream
  - Objects output to ObjectOutputStream
- The difficulty in printing objects is that they may reference other objects
  - How do you make sure that only one object is written and all other refer to it?