Programming in Java

5. Java IO

Streams

- Java uses a basic streams I/O model for most I/O operations
  - Data is accessed through a stream interface
  - Sources are places where data is read from
  - Sinks are places where data is written to
- The streams model is commonly used in many programming languages
  - Meets most of the needs for I/O
  - Random access read/write can be done in Java, but is increasingly rare
  - Most CRUD functionality nowadays is handled by databases and data services instead of flat files
  - Data sources and sinks are wrapped in a stream object
  - Allows a uniform set of operations no matter what physical type the data source or sink is

Stream Types

- There are five basic streams types
- Byte Streams
  - Read or writes a file byte by byte – used for arbitrary data
- Character Streams
  - Reads and writes a file character by character
  - Characters are represented by UTF points
  - Java uses UTF-16, but UTF-8 is now generally the common standard
- Buffered Streams
  - Line oriented reading and writing
  - Standard functionality for reading text files

Stream Types

- Data Streams
  - Manages binary I/O of primitive data types and strings
  - Not covered in this class
- Object Streams
  - Manages the serialization of Java objects

Byte Streams

- Inputs and outputs data in 8-bit chunks
  - Uses the interfaces FileInputStream and FileOutputStream
  - Requires files to be open prior to use
  - Throws IOExceptions if files cannot be accessed
  - This are checked exceptions and must be handled
- The basic read() and write() operations move one byte at a time
  - The read() operations returns a -1 on EOF (end of file)
  - Otherwise it returns the value of the byte it just read

```
try {
    infile = new FileInputStream("SampleText.txt");
    outfile = new FileOutputStream("Copy.txt");

    while ((b = (byte)infile.read()) != -1) {
        outfile.write(b);
        byteCount++;
    }

} catch (IOException e) {
    System.out.println(e);

} finally {
    infile.close();
    outfile.close();

}
```

Byte Array Streams

- A byteArray stream reads a chunk of bytes into a buffer instead of one at a time
  - Disk latency often is the slowest part of reading a file
  - By reading a chunk of data into a memory buffer cuts down on the number of times the disk is accessed
  - The speeds up performance considerably
  - It returns the number of bytes read

```
FileInputStream infile = null;
FileOutputStream outfile = null;

byte[] b = new byte[128];
int inputCount = 0;
int byteCount = 0;
int bytesRead = 0;


while ((bytesRead = infile.read(b)) != -1){
    outfile.write(b);
    byteCount = byteCount + bytesRead;
    inputCount++;
    System.out.println("inputCount=" + inputCount + " bytesRead =" + bytesRead );
}
```

Lab 5-1

Byte Streams

Character Stream

- Inputs and outputs data in single characters
- Uses the interfaces FileReader and FileWriter
  - Manages conversion of bytes to characters
  - The type of text encoding is used to compute how many bytes are needed to read a character
  - The encoding defaults to the whatever the platform default is
  - As of Java 12, the encoding of the files can be specified

Charset	Description
US-ASCII	Seven-bit ASCII, a.k.a. ISO646-US, a.k.a. the Basic Latin block of the Unicode character set
ISO-8859-1	ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1
UTF-8	Eight-bit UCS Transformation Format
UTF-16BE	Sixteen-bit UCS Transformation Format, big-endian byte order
UTF-16LE	Sixteen-bit UCS Transformation Format, little-endian byte order
UTF-16	Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

```
infile = new FileReader("SampleText.txt", StandardCharsets.UTF_8);
outfile = new FileWriter("Copy.txt",StandardCharsets.UTF_8 );
```

Character Array Stream

- Operates analogously to the Byte Array Stream

```
char [] c = new char[128];

try {
    infile = new FileReader("SampleText.txt", StandardCharsets.UTF_8);
    outfile = new FileWriter("Copy.txt", StandardCharsets.UTF_8);

    while ((charsRead = infile.read(c)) != -1) {
        outfile.write(c);
```

Lab 5-2

Character Streams

Buffered Streams

- Java can do buffering so we don’t have to
- The FileReader and FileWriter are wrapped in a BufferedReader and BufferedWriter respectively
  - This is a common idiom in the Java IO library
  - Called the decorator pattern – we take an existing class, FileReader, and add functionality to it, specifically the buffering capability
  - These are generally used for line oriented input
  - The translation of EOL characters is handled automatically;
- When using BufferedWriter
  - The buffer has to be flushed to force a write to the file
  - Otherwise what is in the buffer will not get written to disk

Buffered Streams

```
try {
    infile = new FileReader("SampleText.txt", StandardCharsets.UTF_8);
    inbuff = new BufferedReader(infile);
    outfile = new FileWriter("Copy.txt", StandardCharsets.UTF_8);
    outbuff = new BufferedWriter(outfile);

    while ((line = inbuff.readLine()) != null) {

        outbuff.write(line);
        outbuff.newLine();


    }

} catch (IOException e) {
    System.out.println(e);

} finally {
    outbuff.flush();
    if (inbuff != null) inbuff.close();
    if (outbuff != null) outbuff.close();

}
```

Lab 5-3

Buffered Streams

Serializing Objects

- Java objects are inherently ephemeral
  - They are time bounded – they exist only while the Java program is running
  - They are space bounded – they exist only in the JVM where they were created (specifically on that JVM's memory heap)
- Serialization writes a Java object out to persistent storage
  - This allows the object to be reconstituted later in another Java program
  - It also allows an object to be recreated in another JVM
  - For example, the persistent file can be sent over a network

The Serializable Interface

- Classes that implement the Serializable interface can be save to disk and recovered
  - Serialization writes the object
  - Deserialization recovers the object
- The underlying mechanism of how the process is executed is handled by Java
  - We don’t have to write code to save or recover the object
  - This is all handled by Java
- Although we did not cover data streams
  - Java can write to output data streams all of the primitive data types
  - The same way it does with characters
  - It can also read the primitive types from a data stream correctly.
  - Although it does have to know what type of data it is about to read

Serialization

- Serialization:
  - Saves the instance data
  - Does NOT save static data
  - Does NOT save instance data marked with the transient keyword
- In order to deserialize an object, the JVM must have access to object's class definition
  - The methods of an object are not serialized
  - We usually want to serialize the state of an object which is represented by the instance data
  - If the wrong class definition is being used during deserialization, then an exception is thrown
- It serializes the instance variables by using a data stream
  - If there are instance variables that are user defined classes
  - They also have to implement the Serializable interface
  - Java "walks" the object graph to make sure all parts are serialized

Serial UUID

- In order to ensure proper serialization
  - The class to be serialized has UUID which represents a version of the class
  - This is automatically generated at the time of serialization
  - This is generated from the corresponding ‘.class’ file by a hashing type process
- There are a number of problems with this
  - Different Java versions or platforms can create problems because they might differ in how they internally represent the class files.
  - The complexity of computing the UUID can impact performance
- The alternative is to define our own version ID

Serial UUID

```
class Person implements Serializable {


    private static final long serialVersionUID = 1L;

    private String name = null;
    private int age;
    private transient int id;

    public Person(String name, int age, int id) {
        super();
        this.name = name;
        this.age = age;
        this.id = id;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + ", id=" + id + "]";
    }

}
```

Serialization Output

- The serialization is done by an OutputObjectStream
  - Wraps a FileOutputStream analogous to a BufferedWriter

```
FileOutputStream outfile = new FileOutputStream("person.ser");
ObjectOutputStream out = new ObjectOutputStream(outfile);
out.writeObject(bob);
out.close();
outfile.close();
```

Deserialization Input

- The serialization is done by an InputObjectStream
  - Wraps a FileInputStream analogous to a BufferedReader
  - We must cast the deserialized object to the correct type

```
FileInputStream infile = new FileInputStream("person.ser");
ObjectInputStream in = new ObjectInputStream(infile);
otherBob = (Person) in.readObject();
in.close();
infile.close();
```

Externalizable

- Serialization may not be adequate for some tasks
  - Certain fields may require special handing
  - For example, encryption of credentials
- The Externalizable interface can be used to implement customized serialization
  - Serialization is defined in two methods
  - “writeExternal()” defines how to serialize
  - “readExternal()” defines how to deserialize.

An Externalizable Class

```
public class Country implements Externalizable {

    private String name;
    private int code;

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeUTF(name);
        out.writeInt(code);
    }

    @Override
    public void readExternal(ObjectInput in)
      throws IOException, ClassNotFoundException {
        this.name = in.readUTF();
        this.code = in.readInt();
    }
}
```

Lab 5-4

Serialization


Java™