ECE25100: OBJECT ORIENTED PROGRAMMING

Note 7 _ Streams and Files

Instructor: Xiaoli Yang

Streams and Files

- How to save information from your program to a file.
- How to load that information back in.
- The saving/loading of data from files can be done using different formats. We discuss here the notion of text vs. binary formats.
- The techniques presented here also apply to sending and receiving information from Streams (e.g., networks). We will look at the way in which Stream objects are used to do data I/O in JAVA.

File Terminology

- File processing is very important since eventually, all data must be stored externally from the machine so that it will not be erased when the power is turned off.

- Here are some of the terms related to file processing:

  field

  – A group of characters that conveys meaning. For example, a group of digit characters may represent a number while a group of alphabetic characters may represent a person's name. Note that our objects themselves had fields which were the instance variables.

File Terminology (Cont'd)

record

- A composition of several related fields. For example, a record for an employee may be composed of fields pertaining to employeeNumber, name, address, hourlyRate, ... We can consider all of the state of a particular object instance to be a record.

file

- A group of related records. For example, a company with 20 employees may contain 1 record for each employee.
- Each file ends with an end-of-file (EOF) marker or at a specific byte number recorded in some system data structure.
- To use a file, it must be first opened. When JAVA opens a file, it returns a FileStream object. When done with a file, it MUST be closed.

Database

- A group of possibly unrelated files (e.g., police database containing all criminals, DMV records, phone records)

File Terminology (Cont'd)

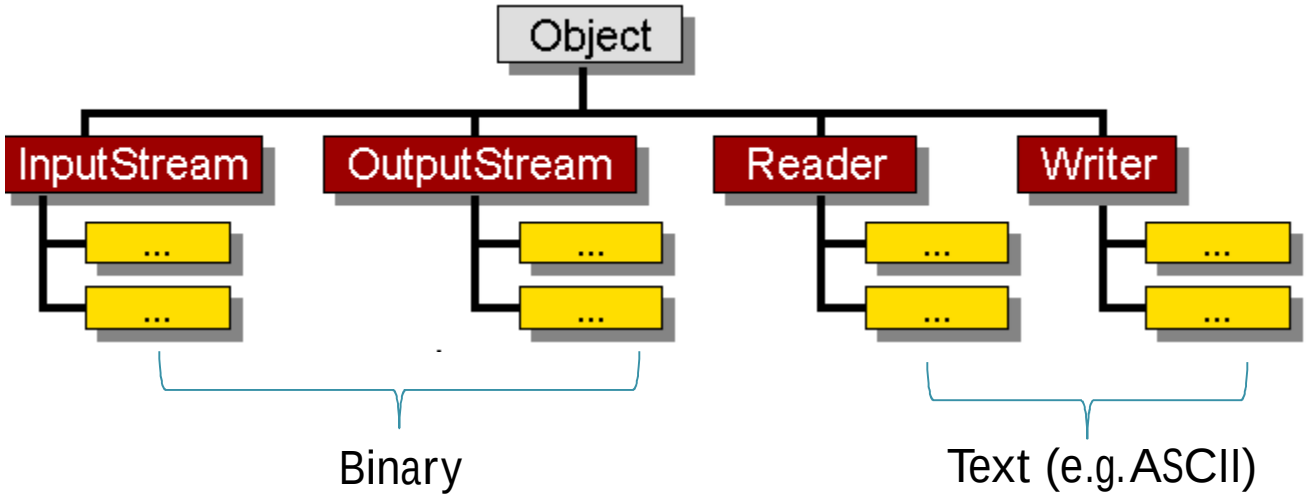
- In JAVA, files are represented as Stream objects. The idea is that data "streams" (or flows) to/from the file … similar to the idea of streaming video that you may have seen online.
- Streams are objects that allow us to send or receive information in the form of bytes. The information that is put into a stream, comes out in the same order.

Streams

- Streams provide a way to send or receive information to and from:
  - files
  - networks
  - different programs
  - any I/O devices (e.g., console and keyboard)
- So you can see that they are not just used for files. We can also create streams that allow us to get data from (or put data into) a "device" other than a file as well. When we first start executing a JAVA program, 3 streams are automatically created:
  - System.in // for inputting data from the keyboard
  - System.out // for outputting data to the standard output location (screen)
  - System.err // for outputting error messages to the screen

Streams (Cont'd)

- There are many stream-related classes in Java. Note that the Streams are all a part of the java.io package.
- Streams in JAVA differ in the way data is "entered into" and "extracted from" the stream. As with exceptions, streams are organized into different hierarchies. JAVA contains four main stream-related hierarchies for transferring data as binary bytes or as text bytes:



Streams (Cont'd)

- Typically I/O is a bottleneck in many applications. It is very time consuming to do I/O operations when compared to internal operations.
- For this reason, buffers are used. Buffered output allows data to be collected for output before it is actually sent to the output device. Only when the buffer gets full does actual data get sent. This allows less output operations to occur by doing few outputs with more data. (Note: The flush() command can be sent to buffered streams in order to empty the buffer and cause the data to be sent "immediately" to the output device. Input data can also be buffered.)

Streams (Cont'd)

- What is System.in and System.out exactly ?
-

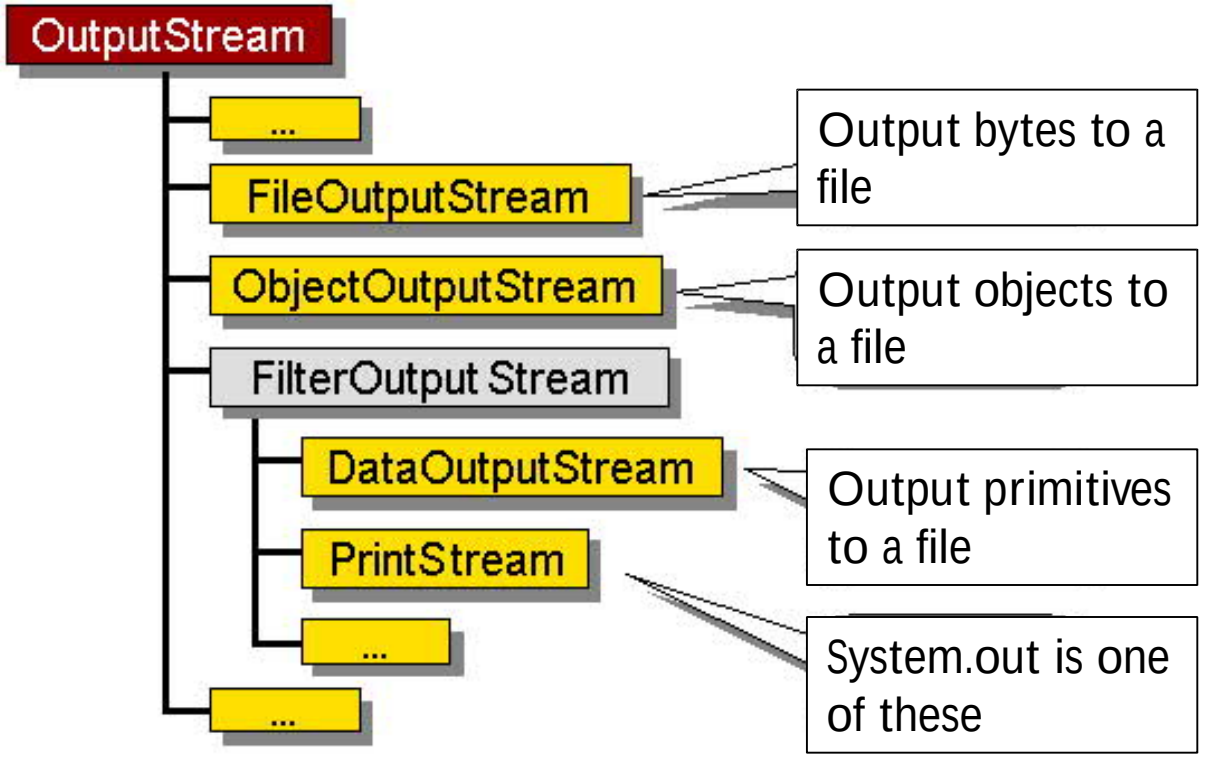
```
System.out.print("System.in is an instance of ");
System.out.println(System.in.getClass());
System.out.print("System.out is an instance of ");
System.out.println(System.out.getClass());
```

- This produces the output:

```
System.in is an instance of class java.io.BufferedInputStream
System.out is an instance of class java.io.PrintStream
```

Reading and Writing Binary Data

- We will now look at how the classes are arranged in the different Stream hierarchies.
- First, let us examine the OutputStream hierarchy:



Reading and Writing Binary Data

- The streams in this hierarchy are responsible for outputting binary data. OutputStreams have a write() method that allows us to output a byte of data at a time.
- To open a file, we usually create an instance of a FileOutputStream using one of the following constructors:

```
new FileOutputStream(String fileName);
new FileOutputStream(String fileName, boolean append);
```

Reading and Writing Binary Data

- The first constructor opens a new file output stream with that name. If one exists already with that name, it is overwritten (i.e., erased). The second constructor allows you to determine whether you want an existing file to be overwritten or appended to.
- If the file already exists prior to this statement,
  - if append = = false, then the existing file's contents is discarded since the file will be overwritten.
  - if append = = true, then the new data to be written to the file is appended to the end of the file.
- If the files does not exist, a new one with that name is created.

Reading and Writing Binary Data

- We can output simple bytes to a file by using a FileOutputStream as follows:

```
FileOutputStream out = new FileOutputStream("myFile.dat");
out.write('H'); out.write(69);
out.write(76);  out.write('L');
out.write('O'); out.write('!');
out.close();
```

- This code outputs the characters HELLO! to the file called "myFile.dat".
- Notice that we can output positive ints (as long as they are less than 256) or characters.
- Notice also that we closed the file when done. This is important to ensure that the operating system releases the file handles.

Reading and Writing Binary Data

- When working with files in this way, two exceptions may occur:
  - opening a file for reading or writing may generate a java.io.FileNotFoundException
  - reading or writing to/from a file may generate a java.io.IOException
- You should handle these exceptions with appropriate try/catch blocks:

Reading and Writing Binary Data

```
import java.io.*;
public class FileOutputStreamTest {
   public static void main(String args[]) {
      try {
         FileOutputStream out = new FileOutputStream("myFile.dat");
         out.write('H'); out.write(69);
         out.write(76);  out.write('L');
         out.write('O'); out.write('!');
         out.close();
      } catch (FileNotFoundException e) {
         System.out.println("Error: Cannot open file for writing.");
      } catch (IOException e) {
         System.out.println("Error: Cannot write to file.");
      }
   }
}
```

Reading and Writing Binary Data

- This code allows us to output any data as long as it is in byte format. This can be tedious. For example, if we have the integer 7293901 and we want to output it, we have a few choices:
  - break up the integer into its 7 digits and output these digits one at a time (very tedious)
  - output the 4 bytes corresponding to the integer itself
- Fortunately, the DataOutputStream class allows us to output whole primitives (e.g., ints, floats, doubles) as well as whole Strings to a file. Here is an example of how to use it to output information from a bank account:

Reading and Writing Binary Data

```
import java.io.*;
public class DataOutputStreamTest {
   public static void main(String args[]) {
      try {
         DataOutputStream out = new DataOutputStream(new
FileOutputStream("myAccount1.dat"));
         BankAccount aBankAccount = new BankAccount(“Rachel”);
         aBankAccount.deposit(100);
         out.writeUTF(aBankAccount.getOwnerName());
         out.writeInt(aBankAccount.getAccountNumber());
         out.writeFloat(aBankAccount.getBalance());
         out.close();
      } catch (FileNotFoundException e) {
         System.out.println("Error: Cannot open file for writing.");
      } catch (IOException e) {
         System.out.println("Error: Cannot write to file.");
      }
   }
}
```

Reading and Writing Binary Data

Notice:

- The DataOutputStream acts as a "wrapper" class around the FileOutputStream. It takes care of breaking our data types and Strings into separate bytes to be sent to the FileOutputStream.
- The methods for writing different data types have different names.

Reading and Writing Binary Data

There are methods to write each of the primitives as well as Strings:

```
writeUTF(String aString)
writeInt(int anInt)
writeFloat(float aFloat)
writeLong(long aLong)
writeDouble(double aDouble)
writeShort(short aShort)
writeBoolean(boolean aBool)
writeByte(int aByte)
writeChar(char aChar)
```

Reading and Writing Binary Data

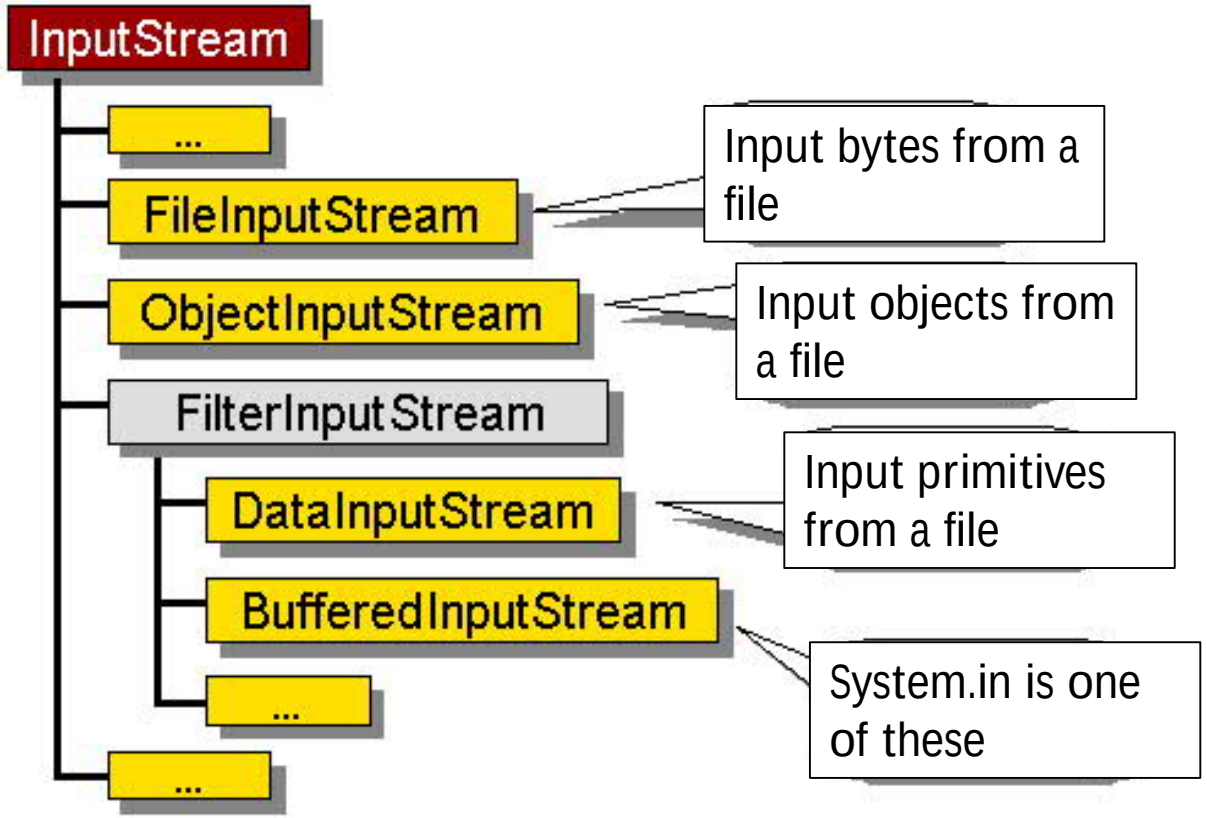
- The output from a DataOutputStream is not very nice to look at (i.e., it is in binary format).
- The myAccount.dat file would display as follows if we tried to view it with Windows Notepad:

  Rachel † BÈ

- This is the binary representation of the data. Different windows programs display the binary data differently.
- So ... we have a way of saving data to a disk ... but can we read it back in again ?

Reading and Writing Binary Data

- We will now examine the InputStream hierarchy:



Reading and Writing Binary Data

- How can we read back in the data from an input files as bytes ? We can use FileInputStream to read data which was outputted using FileOutputStream:
- This code reads back in our file (i.e., the characters HELLO!) and outputs their ASCII (i.e., byte) values to the console:

  72 69 76 76 79 33
- Try changing in.read() to (char)in.read() and see what happens...

Reading and Writing Binary Data

```
import java.io.*;
public class FileInputStreamTest {
   public static void main(String args[]) {
      try {
         FileInputStream in = new FileInputStream("myFile.dat");
        while(in.available() > 0)
            System.out.print(in.read() + " ");
         in.close();
      } catch (FileNotFoundException e) {
         System.out.println("Error: Cannot open file for reading.");
      } catch (EOFException e) {
         System.out.println("Error: EOF encountered, file may be corrupted.");
      } catch (IOException e) {
         System.out.println("Error: Cannot read from file.");
      }
   }
}
```

Reading and Writing Binary Data

- Notice that there is a possibility of an EOFException if there is an attempt to read past the end of the file.
- That was fairly simple ... but what about getting back those primitives ? ?? We'll use DataInputStream:
- Next Slide
- Notice that we re-create a new BankAccount object and "fill-it-in" with the incoming file data.

Reading and Writing Binary Data

```
import java.io.*;
public class DataInputStreamTest {
    public static void main(String args[]) {
        try {
            DataInputStream in = new DataInputStream(new
FileInputStream("myAccount1.dat"));
            BankAccount aBankAccount;
            String name=in.readUTF();
            int acct=in.readInt();
            float bal=in.readFloat();
             aBankAccount=new BankAccount(name,bal,acct);
             System.out.println(aBankAccount);
             in.close();
        } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for reading.");
        } catch (EOFException e) {
            System.out.println("Error: EOF encountered, file may be corrupted.");
        } catch (IOException e) {
            System.out.println("Error: Cannot read from file.");
        }
    }
```

Reading and Writing Binary Data

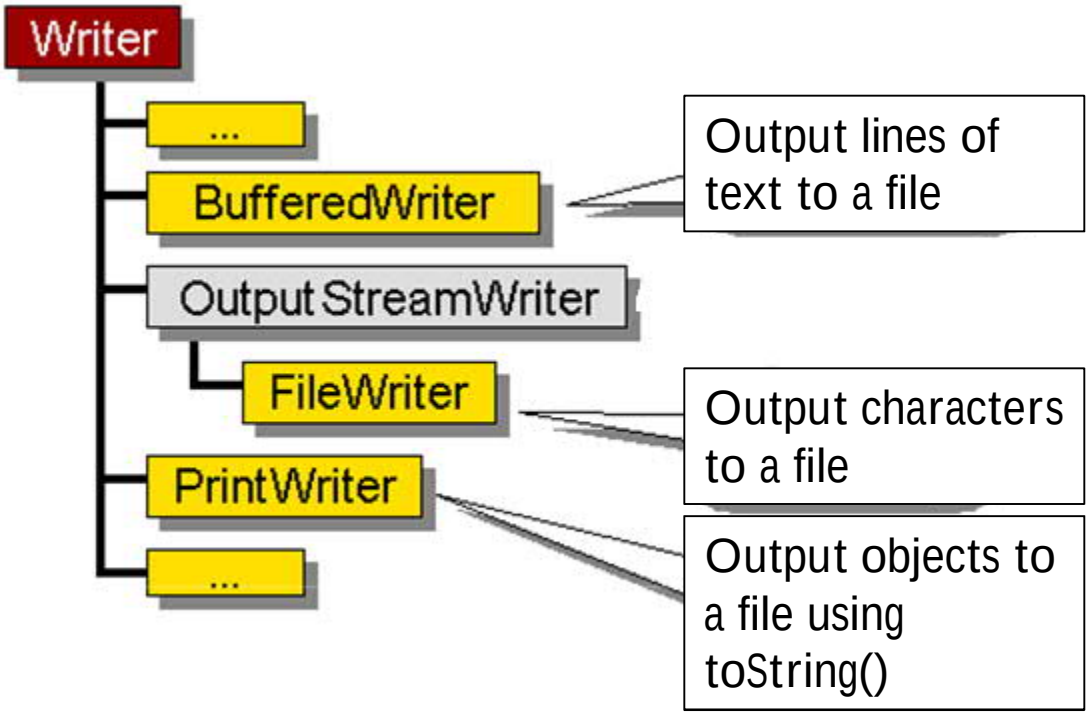
- As with output, there are methods to read the other primitives:

```
String readUTF()
int readInt()
float readFloat()
long readLong()
double readDouble()
 short readShort()
boolean readBoolean()
int readByte()
```

- Also notice that there is a possibility of an EOF Exception if there is an attempt to read past the end of the file. Recall that the order in which you catch your exceptions is important! Catch the most specific ones first. Since IOException is a superclass of EOFException, it must go afterwards.

Reading and Writing Text Data

- We will now examine the Writer hierarchy which is used for writing text to a stream:



Reading and Writing Text Data

- We can output (in text format) to a file using simply the print() or println() methods with the PrintWriter class:

```
import java.io.*;
public class PrintWriterTest {
   public static void main(String args[]) {
      try {
         PrintWriter out = new PrintWriter(new FileWriter("myAccount2.dat"));
         BankAccount aBankAccount = new BankAccount(“Rachel”);
         aBankAccount.deposit(100);
         out.println(aBankAccount.getOwnerName());
         out.println(aBankAccount.getAccountNumber());
         out.println(aBankAccount.getBalance());
         out.close();
      } catch (FileNotFoundException e) {
         System.out.println("Error: Cannot open file for writing.");
      } catch (IOException e) {
         System.out.println("Error: Cannot write to file.");
      }
   }}
```

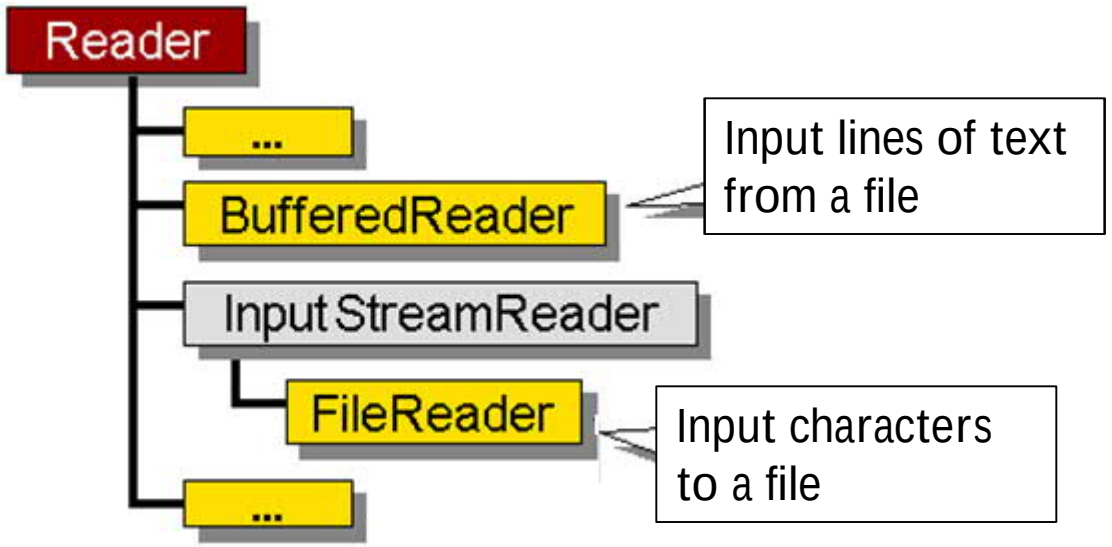
Reading and Writing Text Data

- It almost seems like we are simply outputting to the console window ! But what does it look like ?
- The output is a "pleasant looking" text format:

```
Rachel Class
100000
100.00
```

- In fact, we can actually send any object to the println() method. JAVA will use that Object's toString() method. So it actually does behave just like the System.out console.
- Notice how the PrintWriter wraps the FileWriter class just as the DataOutputStream wrapped the FileOutputStream.
- But how can we read this information back in from the file ?

Reading and Writing Text Data



- We can read buffered input (in text format) from a file using simply the readLine() method:

- We can read buffered input (in text format) from a file using simply the readLine() method:

```
import java.io.*;
public class BufferedReaderTest {
   public static void main(String args[]) {
      try {
         BufferedReader in = new BufferedReader(new
FileReader("myAccount2.dat"));
         BankAccount aBankAccount;
         String name=in.readLine();
         int acct=Integer.parseInt(in.readLine());
         float bal=Float.parseFloat(in.readLine());
         aBankAccount=new BankAccount(name, bal,acc);
         System.out.println(aBankAccount);
         in.close();
      } catch (FileNotFoundException e) {
         System.out.println("Error: Cannot open file for reading.");
      } catch (EOFException e) {
         System.out.println("Error: EOF encountered, file may be corrupted.");
      } catch (IOException e) {
         System.out.println("Error: Cannot read from file.");
      }
   }}
```

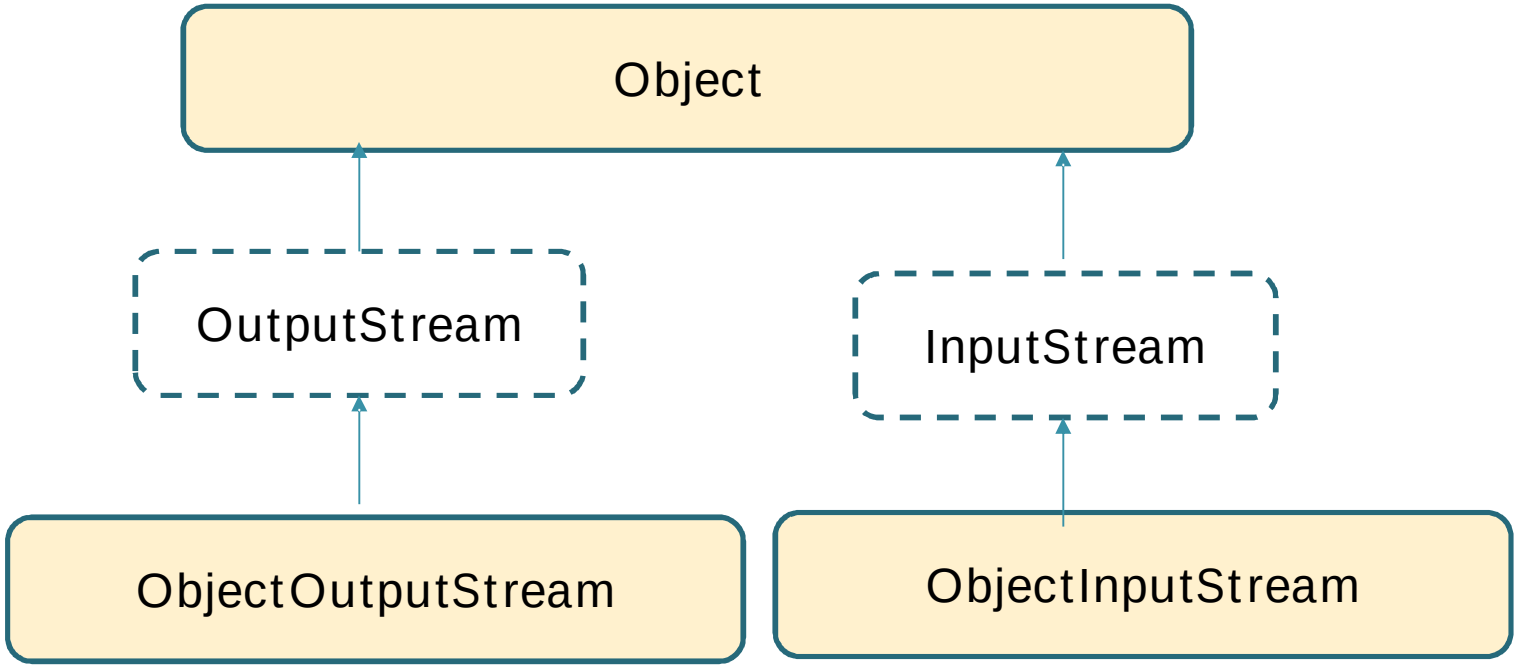
Reading and Writing Text Data

- Note the use of "primitive data type" wrapper classes to read data types. We could have used the Scanner class here to simplify the code.
- Notice here that we now catch a NoSuchElementException. This is how the Scanner detects the end of the file.The main advantage of using this Scanner class is that we do not have to use any wrapper classes to convert the input strings to primitives.

```
import java.io.*;
import java.util.*;
public class BufferedReaderTest2 {
   public static void main(String[] args) {
      try {
            BankAccount aBankAccount;
            Scanner in = new Scanner(new FileReader("myAccount2.dat"));
            String name = in.nextLine();
             int acc = in.nextInt();
             float bal = in.nextFloat();
             aBankAccount = new BankAccount(name, bal, acc);
            System.out.println(aBankAccount);
            in.close();
            } catch (FileNotFoundException e) {
                        System.out.println("Error: Cannot open file for reading");
            } catch (NoSuchElementException e) {
               System.out.println("Error: EOF encountered, file may be corrupt");
             } } }
```

Reading and Writing Whole Objects

- Now we will look at an even simpler way to save/load a whole object to/from a file using the ObjectInputStream and ObjectOutputStream classes:
- These classes allow us to save or load entire Java objects with one method call, , instead of having to break apart the object into its attributes.



Reading and Writing Whole Objects

- Example

```
import java.io.*;
public class ObjectOutputStreamTest {
 public static void main(String args[]) {
    try {
            BankAccount aBankAccount;
            ObjectOutputStream out;
            aBankAccount = new BankAccount(“Rachel");
             aBankAccount.deposit(100);
            out = new ObjectOutputStream(new
FileOutputStream("myAccount3.dat"));
            out.writeObject(aBankAccount);
             out.close();
     } catch (FileNotFoundException e) {
            System.out.println("Error: Cannot open file for writing");
     } catch (IOException e) {
            System.out.println("Error: Cannot write to file");
      }
   }
}
```

Reading and Writing Whole Objects

- We simply supply the object that we want to save to the file as a parameter to the writeObject() method. Notice that the ObjectOutputStream class is a wrapper around the FileOutputStream. That is because ultimately, the object is reduced to a set of bytes by the writeObject() method, which are then saved to the file.
- Serialization is the process of breaking down an object into bytes.
- The JAVA objects can be as complex as you want! The only constraint is that the object must be serializable (i.e., able to be serialized…or reduced to a set of bytes).
- To do this, we need to inform JAVA that our object implements the java.io.Serializable interface as follows.

```
public class BankAccount implements java.io.Serializable { ... }
```

Reading and Writing Whole Objects

- Example

```
import java.io.*;
public class ObjectInputStreamTest {
   public static void main(String[] args) {
      try {
         BankAccount    aBankAccount;
         ObjectInputStream in;
         in = new ObjectInputStream(new FileInputStream("myAccount3.dat"));
         aBankAccount = (BankAccount)in.readObject();

         System.out.println(aBankAccount);
         in.close();
      } catch (ClassNotFoundException e) {
         System.out.println("Error: Object'c class does not match");
      } catch (FileNotFoundException e) {
         System.out.println("Error: Cannot open file for writing");
      } catch (IOException e) {
         System.out.println("Error: Cannot read from file");
      } } }
```

Reading and Writing Whole Objects

- Most standard JAVA classes are serializable by default and so they can be saved/loaded to/from a file in this manner.

```
ArrayList<Customer> customers = new ArrayList<Customer>();
//...add some customers here...
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("customers.dat"));
out.writeObject(customers);
out.close();
```

- All of the types used by this class (and types that use those as wel ... recursively) must implement the interface as well.

```
public class Customer implements java.io.Serializable {
   ... }
```

Reading and Writing Whole Objects

- The object can then be read back in again with the readObject() message. Once read in, the object must be type-casted to the appropriate type:

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("customers.dat"));
ArrayList<Customer> customers = (ArrayList<Customer>)in.readObject();
in.close();
```

Saving and Loading Example

- Lets do a real example now that shows how to save and load the Autoshow.

```
public class Car {
   private String make, model, color;
   private int topSpeed;
   public Car() { this("","","",0); }
   public Car(String mak, String mod, String col, int tsp) {
       make = mak;
       model = mod;
       color = col;
       topSpeed = tsp;
    }
    public String toString() {
    return (make + " " + model + " with top speed " + topSpeed + "mph");
    }
}
```

Saving and Loading Example

```
import java.util.ArrayList;
public class Autoshow {
      private String name;
      private ArrayList<Car> cars;
      public Autoshow(String n) {
            name = n;
            cars = new ArrayList<Car>();
       }
       public void addCar(Car c) {
             cars.add(c);
        }
        public void showCars() {
             for (Car c: cars)
                  System.out.println(c);
         }
}
```

Saving and Loading Example

Chicago Autoshow 2018

Honda
Prelude
White
90

Pontiac
Grand-Am
White
160

Ford
Mustang
White
230

Volkswagon
Rabbit
Blue
100

This seems like a reasonable way to save the autoshow so that the data is readable in a text program. You will notice that each Car object is saved the same way. Hence, it would be good to start by writing some methods that can save/load Car objects.

Saving and Loading Example

- We can write a method in the Car class called saveTo(aFile) which will take a PrintWriter argument:

```
public void saveTo(jPrintWriter aFile) {
    aFile.println(make);
    aFile.println(model);
    aFile.println(color);
    aFile.println(topSpeed);
}
```

Saving and Loading Example

- The method for loading a car from the file is also quite easy. It must create and return a Car object. We'll make it a class method (i.e., static) so that we can call it as follows: Car.loadFrom(aFile)

```
public static Car loadFrom(BufferedReader aFile) {
    Car  newCar = new Car();
    newCar.setName(aFile.readLine());
    newCar.setColor(aFile.readLine());
    newCar.setMake(aFile.readLine());
    newCar.setTopSpeed(Integer.parseInt(aFile.readLine());
    return(newCar);
}
```

- Now we will write some test code to see if it works:

```
import java.io.*;
public class CarSaveLoadTest {
   private static void writeTest() throws IOException {
      PrintWriter          file1, file2;
      Car                  car1, car2;
      file1 = new PrintWriter(new FileWriter("car1.txt"));
      file2 = new PrintWriter(new FileWriter("car2.txt"));
      car1 = new Car("Pontiac", "Grand-Am", "White", 160);
      car2 = new Car("Ford", "Mustang", "White", 230);
      car1.saveTo(file1);
      car2.saveTo(file2);
      file1.close();
      file2.close();
   }
   private static void readTest() throws IOException {
      BufferedReader       file1, file2;
      Car                              car1, car2;
      file1 = new BufferedReader(new FileReader("car1.txt"));
      file2 = new BufferedReader(new FileReader("car2.txt"));
      car1 = Car.loadFrom(file1);
      car2 = Car.loadFrom(file2);
      System.out.println(car1);
      System.out.println(car2);
      file1.close();
      file2.close();    }
   public static void main(String[] args) throws IOException {
      writeTest();
      readTest();    } }
```

Saving and Loading Example

- Now for the fun part. Lets make this work with the Autoshow. We'll make save and load methods as well.

```
public void saveTo(PrintWriter aFile) {
    aFile.println(name);
    //now save the cars
    for (Car c: cars) {
        c.saveTo(aFile);
        aFile.println(); //leave a blank line
    }
}
```

Saving and Loading Example

- The method for loading an Autoshow from the file is also quite easy. It must create and return anAutoshow object. We'll make it a class method so that we can call it as follows: Autoshow.loadFrom(aFile)

```
public static Autoshow loadFrom(BufferedReader aFile) throws IOException {
   Autoshow aShow = new Autoshow(aFile.readLine());

   while (aFile.ready()) { //read until no more available (i.e., not ready)
      aFile.readLine();//read the blank line
      aShow.cars.add(Car.loadFrom(aFile));//read & add the car
    }
    return aShow;
}
```

- Example: AutoShowTest.java

Saving and Loading Example

- Storing Data on a Single Line
  - What if we want to store the Car data on a single line as follows:

```
Honda Prelude White 90
Pontiac Grand-Am White 160
Ford Mustang White 230
Volkswagon Rabbit Blue 100
```

  - Well, saving a Car is easy:

```
public void saveTo2(java.io.PrintWriter aFile) {
   aFile.println(make + " " + model + " " + color + " " + topSpeed);
}
```

Saving and Loading Example

– For reading however, we'll need to alter the load method to use a StringTokenizer to extract the pieces:

```
public static Car loadFrom2(BufferedReader aFile) throws java.io.IOException {
   Car  loadedCar = new Car();
   String[] words = aFile.readLine().split(" ");
   loadedCar.make = words[0];
   loadedCar.model = words[1];
   loadedCar.color = words[2];
   loadedCar.topSpeed = Integer.parseInt(words[3]);
   return loadedCar;
}
```

Saving and Loading Example

- The methods for saving and loading the Autoshow remain the same, except that the blank line should not be written nor read in after each Car.
- But what if the Car's model has two words ?

  Chrysler PT Cruiser Silver 120

- Now its tougher since the name requires two tokens, not one. We can save and load using commas as our delimeters for our tokenizer:

  PT Cruiser,Chrysler,Silver,120

- This solves the problem.

The File Class

- The File class allows us to retrieve information from a file or directory. It has nothing to do with reading and writing to files!!! The class is simply used as a toolkit to get information about files and directories.
- So ... it is a utility class that:
  - has methods for getting file/directory info.
  - cannot be used to read or write to a file.

The File Class

- To make a File object, there are three commonly used constructors:

```
public File(String name);
public File(String pathName, String name);
public File(File directory, String name);
```

- Here are some examples:

```
File file1 = new File("C:\\Data\\myFile.dat");
File file2 = new File("C:\\Data\\", "myFile.dat");
File file3 = new File(new File("."), "myFile.dat");
```

- Notice that filenames may also include full paths as well.

The File Class

- Here are some methods for querying files and directories:

Method	Method	Description
boolean	`canRead();`	**Returns whether or not a file is readable.**
boolean	`canWrite();`	**Returns whether or not a file is writeable.**
boolean	`exists();`	**Returns whether or not a file or directory exists in the specified path.**
boolean	`isFile();`	**Returns whether or not this represents a file.**
boolean	`isDirectory();`	**Returns whether or not this represents a directory.**
boolean	`isAbsolute();`	**Returns whether or not this represents an absolute path to a file or directory.**

The File Class

- Here are other methods for accessing components (i.e., filenames and pathnames) of a file or directory:

Method	Method	Description
`String`	`getAbsolutePath();`	**Return a String with the absolute path of the file or directory.**
`String`	`getName();`	**Return a String with the name of the file or directory.**
`String`	`getPath();`	**Return a String with the path of the file or directory.**
`String`	`getParent();`	**Return a String with the parent directory of the file or directory.**

The File Class

- Here are some other useful methods:

Method	Method	Description
`long`	`length();`	**Return the length of the file in bytes. If the File object is a directory, return 0.**
`long`	`lastModified();`	**Return a system-dependent representation of the time at which the file or directory was last modified. The value returned is only useful for comparison with other values returned by this method.**
`String[]`	`list()`	**Return an array of Strings representing the contents of a directory.**

The File Class

- So here is a program that gives a directory listing (does not go into subdirectories): Check the output.

```
import java.io.*;
public class FileClassTester {
    public static void main(String args[]) {
        File currDir = new File("."); // The dot means the current directory
        System.out.println("The directory name is: " + currDir.getName());
        System.out.println("The path name is: " + currDir.getPath());
        System.out.println("The actual path name is: " + currDir.getAbsolutePath());
        System.out.println("Here are the files in the current directory: ");
        String[]  files = currDir.list();
        for (int i=0; i<files.length; i++) {
           if (new File(files[i]).isDirectory()) System.out.print("***");
            System.out.println(files[i]);
        }
    } }
```

File Naming Conventions

- Path name syntax is different for different platforms, but all JAVA code is meant to run on different platforms. So your file code should be made flexible to work on ALL platforms.
- The File class has a static field called Separator:
  - a string which represents the file separator for the machine that the code is currently running on.
- On a PC, the file separator is the '\' character. In unix or Linux, it is the '/' character. On Macintosh machines, it is yet another character.

File Naming Conventions

- Consider the file BankAccount.java which sits in a directory called models, which itself sits in a directory called FunInc:

```
FunInc
        models
                BankAccount.java
```

- For our code to run on a PC, we can use this hard-coded string:

```
String fileName = "\FunInc\models\BankAccount.java";
```

- In Linus/Unix we can use this:

```
String fileName = "/FunInc/models/BankAccount.java";
```

- Thus, we cannot hard-code the String since they have different separators on different machines.
- Here is the code that we should use to represent the file:

```
String fileName = File.separator + "FunInc" + File.separator +
        "models" + File.separator + "BankAccount.java";
```

References

- Carleton University COMP 1005 and 1006:
  http://www.scs.carleton.ca/~lanthier/teaching/COMP1405/Notes/
  http://www.scs.carleton.ca/~lanthier/teaching/COMP1406/Notes/
- Armstrong Atlantic State University
  http://www.cs.armstrong.edu/liang/intro10e/
- Oracle Java tutorial:
  http://docs.oracle.com/javase/tutorial/
- MIT opencourseware
  https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/