Section 9

Serialization & Writing to Files

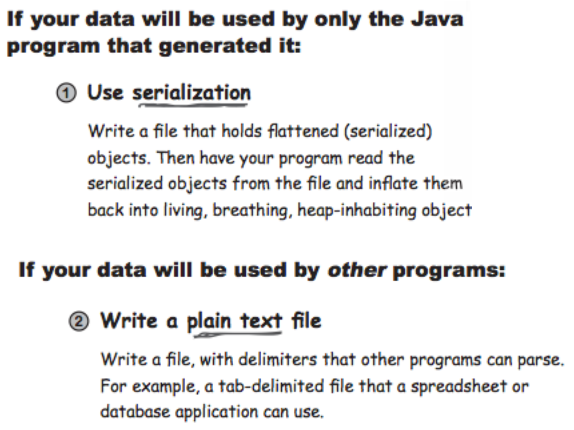
Learning Outcomes

- After this lecture you should be able to:
  - serialize objects
  - Deserialize objects
  - Write to files
  - Read from files

  - ***Reading*** and ***studying*** recommended text is essential to improve your understanding of the above

Saving States

- Need to save the state of your Java program?



If your data will be used by only the Java program that generated it:

① **Use serialization**

Write a file that holds flattened (serialized) objects. Then have your program read the serialized objects from the file and inflate them back into living, breathing, heap-inhabiting object

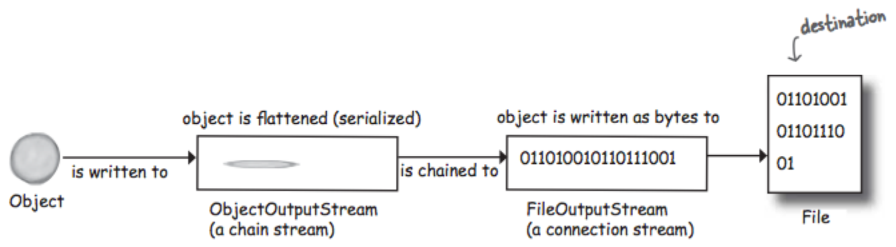
If your data will be used by *other* programs:

② **Write a plain text file**

Write a file, with delimiters that other programs can parse. For example, a tab-delimited file that a spreadsheet or database application can use.

Streams

- Connection streams represent a connection to a source or destination (file, socket, etc.) (Low-level bytes)
- Chain streams cannon connect on their own and must be chained to a connection stream (high-level objects into data)

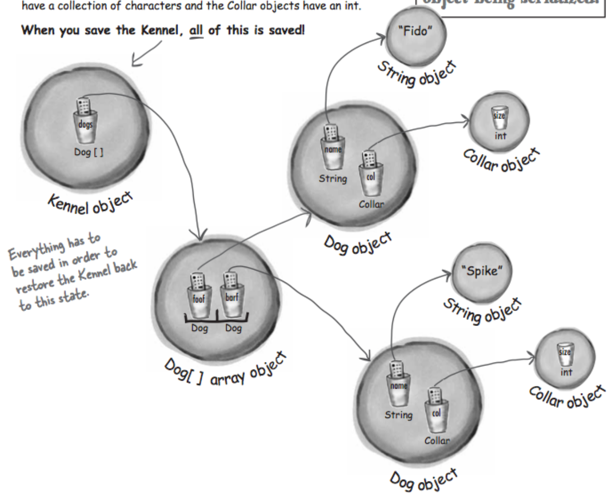


When an object is serialized, all the objects it refers to from instance variables are *also* serialized. And all the objects *those* objects refer to are serialized. And all the objects *those* objects refer to are serialized... and the best part is, it happens automatically!

This Kennel object has a reference to a Dog [] array object. The Dog [] holds references to two Dog objects. Each Dog object holds a reference to a String and a Collar object. The String objects have a collection of characters and the Collar objects have an int.

When you save the Kennel, all of this is saved!

Serialization saves the entire **object graph**. All objects referenced by instance variables, starting with the object being serialized.



serializable

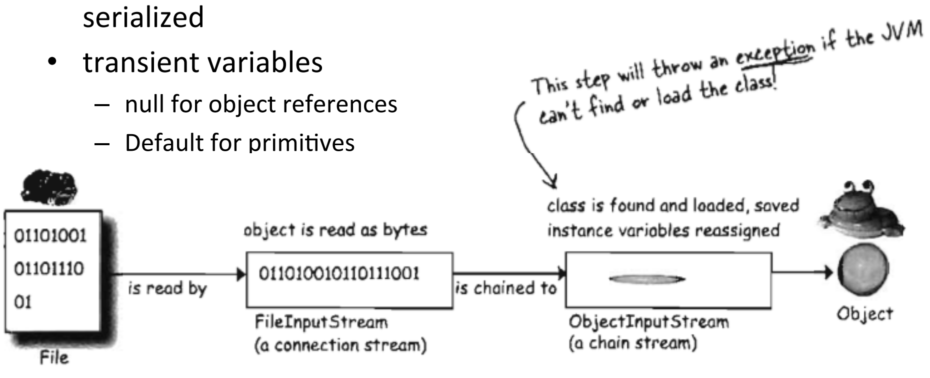
- serializable interface is known as a marker or tag interface as it doesn't have any methods to implement
- Announces that the class implementing it is serializable
- If the superclass "IS-A" serializable, so too is the subclass (interitance)
  - `class Employee implements Serializable`
- Note: serialization is all or nothing i.e. all of the object graph must be serializable.
- If an instance variable can't (or shouldn't) be saved, use the transient keyword
  - `transient String connection;`

Serialization Code

```
try {
        // create a connection stream (write bytes)
        FileOutputStream fileStream = new FileOutputStream("EmployeeInfo.dat");

        // create a chain stream (allows objects to be written to a stream)
        ObjectOutputStream os = new ObjectOutputStream(fileStream);

        // call writeObject() on the Object stream
        os.writeObject(list);

        os.close();

}catch (Exception e) {
        e.printStackTrace();
}
```

Deserialization

- Bring an object back to its original state
- JVM tries to make a new object on the heap that has the same state as the serialized object had at the time it was serialized
- transient variables
  - null for object references
  - Default for primitives



Deserialization Steps

- Object is read from the stream
- JVM determines class type
- JVM finds & loads object's classes – throws an exception
- New object is given space on the heap (serialized objects constructor does NOT run)
- If the object has a non-serializable class in its inheritance tree, the constructor for that non-serializable class will run (constructor chaining begins!)
- Object's instance variables are given the values from the serialized state (transient variables are given default values)

Deserialize Code

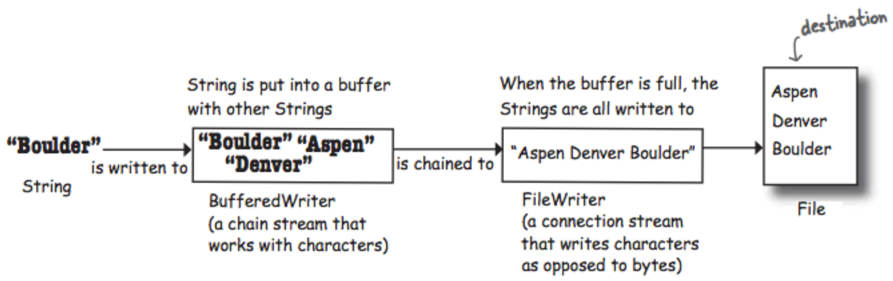
```
try{
          FileInputStream fileStream = new FileInputStream("EmployeeInfo.dat");

          ObjectInputStream os = new ObjectInputStream(fileStream);

          List<Employee> emp = (List<Employee>)os.readObject();

          for (Employee element:emp){
                    System.out.println(element.toString());
          }

          os.close();

          } catch (Exception e) {
                    e.printStackTrace();
          }
```

Writing to Files

- Saving objects through serialization is the easiest way to save & restore data between runnings of a Java program.
- However, you may need to save data to a text file
- Writing text data (String) is similar to writing an object, except you use FileWriter instead of a FileOutputStream and you don't chain it to an ObjectOutputStream.

Buffers

- Buffers make writing more efficient
- Offer a temporary holding place to group things until the buffer is full.
- Once full writing commences


"Boulder"
String
is written to
String is put into a buffer with other Strings
"Boulder" "Aspen" "Denver"
BufferedWriter (a chain stream that works with characters)
is chained to
When the buffer is full, the Strings are all written to
"Aspen Denver Boulder"
FileWriter (a connection stream that writes characters as opposed to bytes)
destination
Aspen
Denver
Boulder
File

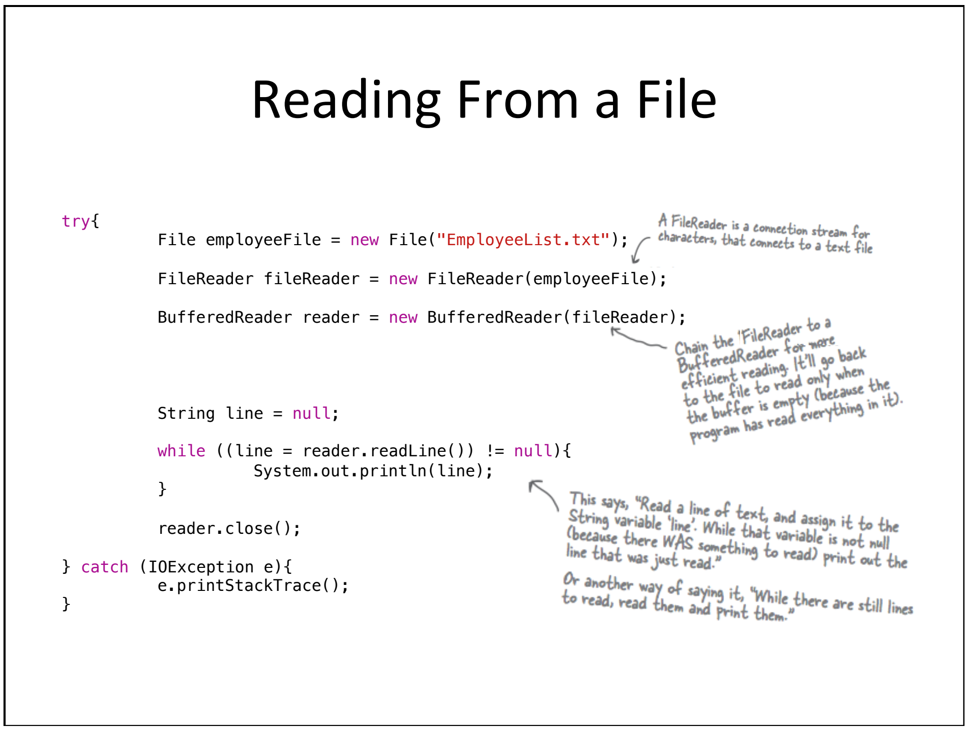
Writing to a File

```
try {

    FileWriter fileWriter = new FileWriter("EmployeeList.txt");

    BufferedWriter writer = new BufferedWriter(fileWriter);

    writer.write("Employee List \n");

          for(Employee element:list){
              writer.write("Name:  " + element.getName() + " \n ");
              writer.write("Employee Number: " + element.idNumber  + " \n ");
              writer.write("PPS Number: " + element.getPPS() + " \n ");

              if (element instanceof HourlyEmployee){
                    writer.write("Hours Worked: " + Double.toString(((HourlyEmployee)
element).getHoursWorked()) + " \n ");
              }
              else {
                  writer.write("Commission " + Double.toString(((CommissionEmployee)
element).getSales()) + " \n ");
              }
              writer.write("Pay € " + (Double.toString(element.pay)) + " \n ");
                                        writer.write("\n");
          }
    writer.close();
} catch(IOException e){
      e.printStackTrace();
}
```

Reading From a Text File

- Use a File object to represent a file and a FileReader to do the reading and use a BuferedReader to make the reading more efficient

- Read happens by reading lines in with a *while* loop, which ends when the result of a readLine() is null



Reading From a File

```
try{
          File employeeFile = new File("EmployeeList.txt");

          FileReader fileReader = new FileReader(employeeFile);

          BufferedReader reader = new BufferedReader(fileReader);



          String line = null;

          while ((line = reader.readLine()) != null){
                    System.out.println(line);
          }

          reader.close();

} catch (IOException e){
          e.printStackTrace();
}
```

A FileReader is a connection stream for characters, that connects to a text file

Chain the FileReader to a BufferedReader for more efficient reading. It'll go back to the file to read only when the buffer is empty (because the program has read everything in it).

This says, "Read a line of text, and assign it to the String variable 'line'. While that variable is not null (because there WAS something to read) print out the line that was just read."

Or another way of saying it, "While there are still lines to read, read them and print them."