



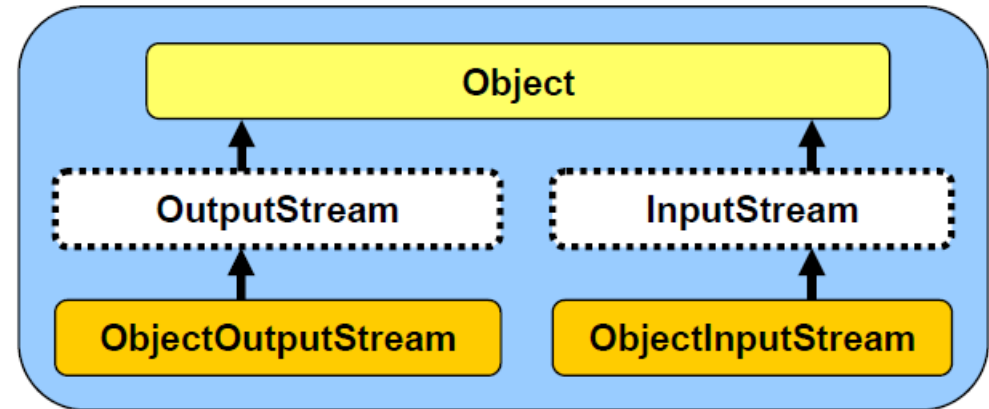
# Application Program Development

Segment : Java Files and Streams

Mahboob Ali

# Reading and Writing Whole Objects

- So far, we have seen ways of saving and loading bytes and characters to a file.
- Also, we have seen how **DataOutputStream/DataInputStream** and **PrintWriter/BufferedReader** classes can make our life simpler since they deal with larger (more manageable) chunks of data such as primitives and Strings.
- We also looked at how we can save a whole object (i.e., a **BankAccount**) to a file by extracting its attributes and saving them individually.
- 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



# Reading and Writing Whole Objects

- 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.

```
BankAccount aBankAccount;  
ObjectOutputStream out;  
aBankAccount = new BankAccount("Rob Banks");  
aBankAccount.deposit(100);  
out = new ObjectOutputStream(new FileOutputStream("myAcc.dat"));  
out.writeObject(aBankAccount);  
out.close();
```

- It is VERY easy to write out an object.
- 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.

# Reading and Writing Whole Objects → Serialization

- **Serialization** is the process of breaking down an object into bytes.
- Thus, when an object is saved to a file, it is automatically de-constructed into bytes, these bytes are then saved to a file, and then the bytes are read back in later and the object is re-constructed again.
- This is all done automatically by JAVA, so we don't have to be too concerned about it.
- In order to be able to save an object to a file using the **ObjectOutputStream**, 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 {  
    ...  
}
```

- This particular interface does not actually have any methods within it that we need to implement.
- Instead, it merely acts as a "flag" that indicates your permission for this object to be serialized.
- It allows a measure of security for our objects (i.e., only **serializable** objects are able to be broken down into bytes and sent to files or over the network).

# Reading and Writing Whole Objects → Serialization

- Most standard JAVA classes are **serializable** by default and so they can be saved/loaded to/from a file in this manner. When allowing our own objects to be serialized, we must make sure that all of the “pieces” of the object are also **serializable**.
- For example, assume that our **BankAccount** is defined as follows:

```
public class BankAccount implements java.io.Serializable {  
    Customer owner;  
    float balance;  
    int accountNumber;  
    ...  
}
```

- In this case, since owner is not a **String** but a **Customer** object, then we must make sure that **Customer** is also **Serializable**:

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

- We would need to then check whether **Customer** itself uses other objects and ensure that they too are **serializable** ... and so on.
- To understand this, just think of a meat grinder. If some marbles were placed within the meat, we cannot expect it to come out through the grinder since they cannot be reduced to a smaller form.
- Similarly, if we have any non-**serializable** objects in our original object, we cannot properly serialize the object.

# Reading and Writing Whole Objects → Serialization

- So what does a serialized object look like anyway ?
- Here is what the file would look like from our previous example if opened in Windows Notepad:  

```
í sr BankAccount"ÈSòñúä I accountNumberF balanceL ownert Ljava/lang/String;xp † BÈ t Rob Banks
```
- Weird ... it seems to be a mix of binary and text.
- As it turns out, JAVA saves all the attribute information for the object, including their types and values, as well as some other information.
- It does this in order to be able to re-create the object when it is read back in.

# Reading and Writing Whole Objects → De-Serialization

- The object can be read back in by using the **readObject()** method in the **ObjectInputStream** class as follows

```
try{
    BankAccount aBankAccount;
    ObjectInputStream in;
    in = new ObjectInputStream(new FileInputStream("myAcc.dat"));
    aBankAccount = (BankAccount) in.readObject();
    System.out.println(aBankAccount);
    in.close();
} catch (ClassNotFoundException e) { System.out.println("Error: Object's 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"); }
```

- Note, that the **ObjectInputStream** wraps the **FileInputStream**.
- Also, notice that once read in, the object must be *type-casted* to the appropriate type (in this case **BankAccount**).
- Also, if there is any problem trying to re-create the object according to the type of object that we are loading, then a **ClassNotFoundException** may be generated, so we have to handle it.
- Finally, in order for this to work, you must also make sure that your object (i.e., **BankAccount**) has a zero-parameter constructor, otherwise an **IOException** will occur when JAVA tries to rebuild the object.
- Although not shown in our example, you may also make use of the **available()** method to determine whether or not the end of the file has been reached.

# Reading and Writing Whole Objects → De-Serialization

- Although this method is extremely easy to use, there is a potentially disastrous disadvantage.
- The object that is saved to the file using this strategy is actually saved in binary format which depends on the class name, the object's attribute types and names as well as the method signatures and their names.
- So if you change the class definition after it has been saved to the file, it may not be able to be read back in again !!! Some changes to the class do not cause problems such as adding an attribute or changing its access modifiers.
- So as a warning, when saving objects to a file using this strategy, you should always keep a backed-up version of all of your code so that you will be able to read these files with this backed-up code in the future.



# Disguising Serialized Data

- You can actually write your own methods for serializing your objects.
- One reason for doing this may be to encrypt some information beforehand (such as a password).
- You can decide which parts of the object will be serialized and which parts will not.
- You can declare any object attribute as being ***transient*** (which means that it will not be serialized) as follows:  
    private transient String password;
- This will tell JAVA that you do not want the password saved automatically upon serialization.
- That way you can write your own method to encrypt it before it is serialized.
- To do this, you would need to write two methods called **writeObject(ObjectOutputStream)** and **readObject(ObjectInputStream)**.
- These methods will automatically be called by JAVA upon serialization and they override the default writing behavior.
- In fact, there are **defaultWriteObject()** and **defaultReadObject()** methods which do the default serialization behavior (i.e., the serializing before you decided to do your own).

# Disguising Serialized Data

- Here are examples of what you can do:



```
public void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject();


    // ... do extra stuff here to append to end of file
    out.writeObject(myField.encrypt());
}

public void readObject(ObjectInputStream in) throws IOException,
                                                                ClassNotFoundException {

    in.defaultReadObject();

    // ... do extra stuff here to read from end of file
    myField = ((myFieldType)in.readObject()).decrypt();
}
```

- 
1. If a parent class has implemented Serializable interface then child class doesn't need to implement it but vice-versa is not true.
  2. Only non-static data members are saved via Serialization process.
  3. Static data members and transient data members are not saved via Serialization process. So, if you don't want to save value of a non-static data member then make it transient.
  4. Constructor of object is never called when an object is de-serialized.
  5. Associated objects must be implementing Serializable interface.
- 

- 
- All of the streams you have used so far are known as *read-only* or *write-only* streams.
  - The external files of these streams are *sequential* files that cannot be updated without creating a new file.
  - It is often necessary to modify files or to insert new records into files.
  - Java provides the **RandomAccessFile** class to allow a file to be read from and write to at random locations.

- A random access file consists of a sequence of bytes.
- *File pointer*: a special marker that is positioned at one of these bytes.
- A read or write operation takes place at the location of the file pointer.
- When a file is opened, the file pointer sets at the beginning of the file.
- When you read or write data to the file, the file pointer moves forward to the next data.
- For example, if you read an int value using `readInt()`, the JVM reads four bytes from the file pointer and now the file pointer is four bytes ahead of the previous location.

- Many methods in *RandomAccessFile* are the same as those in *DataInputStream* and *DataOutputStream*.
- For example:
  - `readInt()`,
  - `readLong()`,
  - `writeDouble()`,
  - `readLine()`,
  - `writeInt()`,
  - `writeLong()`.
- `void seek(long pos) throws IOException;`  
Sets the offset from the beginning of the *RandomAccessFile* stream to where the next read or write occurs.

- `long getFilePointer() IOException;`  
Returns the current offset, in bytes, from the beginning of the file to where the next read or write occurs.
- `long length() IOException`  
Returns the length of the file.
- `final void writeChar(int v) throws IOException`  
Writes a character to the file as a two-byte Unicode, with the high byte written first.
- `final void writeChars(String s) throws IOException`  
Writes a string to the file as a sequence of characters.

- `RandomAccessFile raf = new RandomAccessFile("test.dat", "rw");`
- `// allows read and write`
- `RandomAccessFile raf = new RandomAccessFile("test.dat", "r");`
- `// read only`