8 - I/O Fundamentals

< Previous | Next >

§8.1 Character and Binary Streams

- Methods of the InputStreamReader:
  - The ready method just checks if there are more bytes available to read.
  - The skip method skips the given number of characters i.e. it basically moves the file pointer one character ahead.
  - The read method reads one character.

- `FileWriter fw = new FileWriter("text.txt");` If the file does not exist, it will be created and data will be written to it when you call the write method. If nothing is written to the file, it will create an empty file. If the file already exists, it will be overwritten with a new file. To append to the existing file, the following constructor should be used. `public FileWriter(String fileName,  boolean append)` .
- You should remember that following points:
  1. BufferedWriter only adds the functionality of buffering on top of a Writer. It doesn't directly deal with encoding. Encoding is handled by the underlying Writer object.
  2. FileWriter is a concrete subclass of java.io.Writer that writes data to the underlying file in default encoding. If you want to write text in a different encoding, you will need to create an OutputStreamWriter with that encoding. For example,
     `OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("utf8.txt"), Charset.forName("UTF-8").newEncoder() );`
     You can then create a BufferedWriter over this OutputStreamWriter.

- `InputStream.read()` fills the buffer with the bytes actually read. So even if the buffer is larger than the available number of bytes, it is not a cause for any exception.
- When you create a FileOutputStream without specifying the append mode (which can be true or false), it overwrites the
- When you create a FileOutputStream without specifying the append mode (which can be true or false), it overwrites the existing file.
- While StringReader is a valid class but it creates a Reader out of a String. It does not read Strings from a Reader. For example:
  `StringReader sr = new StringReader("some long string");`
- Note that the write(int b) method of various streams based classes such as FileOutputStream take an int parameter but write only the low 8 bits (i.e. 1 byte) of that integer.
- DataOutputStream provides methods such as writeInt, writeChar, and writeDouble, for writing complete value of the primitives to a file. So if you want to write an integer to the file, you should use writeInt(1) in which case a file of size 4 bytes will be created. You can read back the stored primitives using methods such as DataInputSream.readInt().
- OutputStream does not provide methods for writing primitives. It writes bytes only.
- `java.io.FileNotFoundException` may be thrown by FileInputStream, FileOutputStream, and RandomAccessFile constructors if the file by the given name does not exist.
- `java.nio.file.NoSuchFileException` will be thrown when the program tries to create a BufferedReader to read the file specified by the Path object.
- `java.nio.file.InvalidPathException` is thrown when the argument passed while creating a Path object is invalid. For example, "c:c:test.txt". The existence of the file is not checked at the time of creation of Path object.
- `File f = new File("path-to-file"");`
  - `f.getParent()` returns a String.
  - `f.getParentFile()` returns a File object.
  - `f.mkdirs()` Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories. Returns: true if and only if the directory was created, along with all necessary parent directories; false otherwise.
  - `f.mkdir()` can only create the last component of a path. It cannot create a directory structure.

- A BufferedReader can only be created using a Reader such as FileReader. It cannot directly operate on a file.
- Reader does not have high level methods such as readLine().
- A Reader such as a FileReader provides only low level operations such as reading a single character or array of characters. It does not understand the notion of "lines". BufferedReader "decorates" Reader to provide higher level method readLine() by buffering characters. It is an efficient way of reading characters, character arrays, and lines. The same relationship exists between FileWriter and BufferedWriter but for writing.

 # BufferedWriter:

- BufferedWriter does not have writeUTF method but it does have newLine and write(String) methods.
- A Reader can be chained to a BufferedReader to read Strings. BufferedReader has readLine method that returns a String.
- There is no readLines method in BufferedReader. There is a readLine method but it returns only one line.
- BufferedWriter's append method works same as the write(String) method. It doesn't really append the data to the end of the existing content. It overwrites the existing content.
- The flush method flushes the stream and makes sure any data that is in the stream but is not written to the file yet, is written to the file. It does not close the stream. A call to flush is useful when you want to write the contents to the file but don't want to close the writer yet.
- The close method flushes the stream and makes sure that all data is actually written to the file.

 # PrintWriter:

- PrintWriter does not provide explicit methods for writing primitives (i.e. writeInt, writeBoolean, etc.). It has overloaded **print** methods that take various primitives (i.e. print(int), print(boolean), print(long), and print(char) as arguments).
- PrintWriter does not have write(boolean) method. It does have write(String), write(int), write(char[]) methods. It also has write(char[] buf, int off, int len) and write(String buf, int off, int len) methods that let you write a portion of the input buf.
- PrintWriter's write method writes a single character to the file. The size in bytes of a character depends on the default character encoding of the underlying platform. For example, if the encoding is UTF-8, only 1 byte will be written and the size of the file will be 1 byte.
- **Remember that none of PrintWriter's print or write methods throw I/O exceptions** (although some of its constructors may). This is unlike other streams, where you need to include exception handling (i.e. a try/catch or throws clause) when you use the stream.
- Note that none of the PrintWriter's methods throw any I/O exceptions because they supress the errors in writing and set an internal flag for error status instead. The checkError method returns true if there has been a problem in writing.
- PrintWriter has `printf(Locale l, String format, Object... args)` and `printf(String format, Object... args)` methods that allow you to format the input before printing. These methods return the same PrintWriter object so that you can chain multiple calls as shown in this option.
- All the write and print methods (except for printf) of PrintWriter return void. No chaining!
- Note that the println methods of PrintWriter cause the data written in the stream to be flushed if automatic flushing is enabled. You can set the autoflush behaviour by using `PrintWriter(OutputStream out, boolean autoFlush)` constructor while creating a PrintWriter.

§8.2 Serialization

- The no-argument constructor of only the first non-serializable super class is invoked. This constructor may internally invoke any constructor of its super class.
- The same object can be serialized multiple times to a stream. Thus, you may have mutliple copies of the same object in the stream and when you read the stream back, you will get multiple objects.
- Developers can change how a particular class is serialized by implementing two methods inside the class file. These methods are:
  `private void writeObject(ObjectOutputStream out) throws IOException;`
  `private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;` Notice that both methods are (and must be) declared private, proving that neither method is inherited and overridden or overloaded. The trick here is that the virtual machine will automatically check to see if either method is declared during the corresponding method call. The virtual machine can call private methods of your class whenever it wants but no other objects can. Thus, the integrity of the class is maintained and the serialization protocol can continue to work as normal.
- If the object graph contains non-serializable objects, an exception is thrown and nothing is serialized. Object graph means all the objects that are linked/referenced by the first object (directly or indirectly) that is being serialized. Fields of an Object that are marked as transient are not serialized and so they do not cause an exception. Any field that is not marked transient but points to an object of a class that does not implement Serializable, will cause an exception to be thrown. Thus, the given statement is wrong.
- If a serializable class contains a reference to a non-serializable object, but you want to preserve the state of these objects, you have two options:
  1. make the objects implement Serializable
  2. make the reference transient and implement readObject(ObjectInputStream os) and writeObject(ObjectOutputStream os) methods to read and write the state of the objects explicitly.

- Remember that transient fields and static fields are never serialized. Constructor, instance blocks, and field initialization of the class being deserialized are also not invoked.
- Remember that static fields are never serialized irrespective of whether they are marked transient or not. In fact, making static fields as transient is redundant. However, since f4 `public final static String f4 = "4";` is being initialized as a part of class initialization, it will be initialized to the same value in another JVM. Thus, its value will be same as the one initialized by the code.

```
public class Portfolio implements Serializable {
    String accountName; transient Bond bond = new Bond();

    private void writeObject(ObjectOutputStream os){
        os.defaultWriteObject();
        os.writeObject(bond.ticker);
        os.writeObject(bond.coupon);
    }

    private void readObject(ObjectInputStream is){
        is.defaultReadObject();
        this.bond = new Bond();
        bond.ticker = (String) is.readObject();
        bond.coupon = is.readDouble();
    }
}
```

1. Bond class does not implement Serializable. Therefore, for Portfolio to be serialized, 'bond' must be made transient.
2. writeObject method takes ObjectOutputStream as the only parameter, while readObject method takes ObjectInputStream.
3. To serialize the object using the default behavior, you must call `objectOutputStream.defaultWriteObject();` or `objectOutputStream.writeFields();`. This will ensure that instance fields of Portfolio object are serialized.
4. To deserialize the object using the default behavior, you must call `objectInputStream.defaultReadObject();` or `objectInputStream.readFields();`. This will ensure that instance fields of Portfolio object are deserialized.

5. The order of values to be read explicitly in readObject must be exactly the same as the order they were written in writeObject. Here, ticker was written before coupon and so ticker must be read before coupon.

§8.3 Console

- Whether a virtual machine has a console is dependent upon the underlying platform and also upon the manner in which the virtual machine is invoked. If the virtual machine is started from an interactive command line without redirecting the standard input and output streams then its console will exist and will typically be connected to the keyboard and display from which the virtual machine was launched. If the virtual machine is started automatically, for example by a background job scheduler, then it will typically not have a console.
- If this virtual machine has a console then it is represented by a unique instance of this class which can be obtained by invoking the System.console() method. If no console device is available then an invocation of that method will return null.
- Console is meant to interact with the user typically through command/shell window and keyboard. Thus, binary data doesn't make sense for the console. You can read whatever the user types using `readLine()` and `readPassword()` method. You can also acquire a Reader object using `reader()` method on Console object. All these provide character data. Similarly, you can acquire PrintWriter object using `writer()` method on Console, which allows you to write character data to the console.
- None of the calls to Console throw any checked exception. Call to System.console() doesn't throw any exception either. It just returns null if Console is not available.

 # Available methods:

- `public PrintWriter writer()` Retrieves the unique PrintWriter object associated with this console.
- `public Reader reader()` Retrieves the unique Reader object associated with this console.
- `public Console format(String fmt, Object... args)` Writes a formatted string to this console's output stream using the specified format string and arguments.
- `public Console printf(String format, Object... args)` A convenience method to write a formatted string to this console's output stream using the specified format string and arguments.
- `public String readLine()` Reads a single line of text from the console.
- `public String readLine(String fmt, Object... args)` Provides a formatted prompt, then reads a single line of text from the console.
- `public char[] readPassword()` Reads a password or passphrase from the console with echoing disabled.
- `public char[] readPassword(String fmt, Object... args)` Provides a formatted prompt, then reads a password or passphrase from the console with echoing disabled.
- `public void flush()` Flushes the console and forces any buffered output to be written immediately.

< Previous | Next >