Chapter 5

I/O and NIO

Business Training

Content

- File Navigation and I/O
- Files, Path, and Paths
- Serialization

File Navigation and I/O

Overview of I/O Classes

- Here's a summary of the I/O classes you'll need to understand for the exam:
  - File
  - FileReader
  - BufferedReader
  - FileWriter
  - BufferedWriter
  - PrintWriter
  - FileInputStream
  - FileOutputStream
  - ObjectInputStream
  - ObjectOutputStream
  - Console

Creating Files Using the File Class

- **Objects of type File are used to represent the actual files (but not the data in the files) or directories that exist on a computer's physical disk.**
  - Example:

```
import java.io.*;

class Writer1 {
 public static void main(String [] args) {
   try {                                  // warning: exceptions possible
     boolean newFile = false;
     File file = new File                 // it's only an object
                 ("fileWrite1.txt");
     System.out.println(file.exists());   // look for a real file
     newFile = file.createNewFile();      // maybe create a file!
     System.out.println(newFile);         // already there?
     System.out.println(file.exists());   // look again
   } catch(IOException e) { }
 }
}
```

  - This produces the output: **false, true, true**
  - And also produces an empty file in your current directory. If you run the code a *second* time, you get the output : **true, false, true**

Using FileInputStream and FileOutputStream

- **Using FileInputStream and FileOutputStream is similar to using FileReader and FileWriter, except you're working with byte data instead of character data.**
  - That means you can use FileInputStream and FileOutputStream to read and write binary data as well as text data.
- **Example**

```
import java.io.*;
class Writer3 {
   public static void main(String [] args) {
      byte[] in = new byte[50];              // bytes, not chars!
      int size = 0;
      FileOutputStream fos = null;
      FileInputStream fis = null;
      File file = new File("fileWrite3.txt");
      try {
         fos = new FileOutputStream(file);   // create a FileOutputStream
         String s = "howdy\nfolks\n";
         fos.write(s.getBytes("UTF-8"));     // write characters (bytes)
                                             // to the file
         fos.flush();                        // flush before closing
         fos.close();                        // close file when done

         fis = new FileInputStream(file);    // create a FileInputStream
         size = fis.read(in);                // read the file into in
         System.out.print(size + " ");       // how many bytes read
         for(byte b : in) {                  // print the array
            System.out.print((char)b);
         }
         fis.close();                        // again, always close
      } catch(IOException e) {
         e.printStackTrace();
      }
   }
}
```

java.io Mini API

java.io Class	Extends From	Key Constructor(s) Arguments	Key Methods
File	Object	File, String String String, String	createNewFile() delete() exists() isDirectory() isFile() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
BufferedWriter	Writer	Writer	close() flush() newLine() write()
PrintWriter	Writer	File (as of Java 5) String (as of Java 5) OutputStream Writer	close() flush() format(), printf() print(), println() write()
FileOutputStream	OutputStream	File String	close() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()
FileInputStream	InputStream	File String	read() close()

The java.io.Console Class

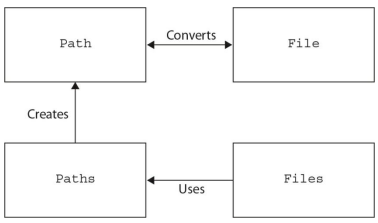
- **Java 6 added the java.io.Console class. In this context, the *console* is the physical device with a keyboard and a display (like your Mac or PC).**
  - Let's take a look at a small program that uses a console to support testing another class:

```
import java.io.Console;

public class NewConsole {
  public static void main(String[] args) {
    String name = "";
    Console c = System.console();              // #1: get a Console
    char[] pw;
    pw = c.readPassword("%s", "pw: ");         // #2: return a char[]
    for(char ch: pw)
      c.format("%c ", ch);                     // #3: format output
    c.format("\n");

    MyUtility mu = new MyUtility();
    while(true) {
      name = c.readLine("%s", "input?: ");     // #4: return a String

      c.format("output: %s \n", mu.doStuff(name));
    }
  }
}
```

Files, Path, and Paths

Java File I/O (NIO.2)

- **The term *NIO.2* is a bit loosely defined, but most people (and the exam creators) define NIO.2 as being the key new features introduced in Java 7 that reside in two packages:**
  - java.nio.file
  - java.nio.file.attribute
- **NIO.2 adds three new central classes that you'll need to understand well for the exam:**
  - **Path** This interface replaces File as the representation of a file or a directory when working in NIO.2. It is a lot more powerful than a File.
  - **Paths** This class contains static methods that create Path objects.
  - **Files** This class contains static methods that work with Path objects. You'll find basic operations in here like copying or deleting files.
- **Let's take a look at these relationships another way.**
  - The Paths class is used to create a class implementing the Path interface.
  - The Files class uses Path objects as parameters.



Creating/Normalizing a Path

- **A Path object can be easily created by using the get methods from the Paths helper class.**
  - Remember you are calling Paths.get() and not Path.get()

```
Path p1 = Paths.get("/tmp/file1.txt");    // on UNIX
Path p2 = Paths.get("c:\\temp\\test");    // On Windows
```

  - **Paths.get(URI uri)** lets you (indirectly) convert the String to a URI (Uniform Resource Identifier) before trying to create a Path:

```
Path p = Paths.get(URI.create("file:///C:/temp"));
```

- **Normalize() simplifies path representation**

```
System.out.println(Paths.get("/a/./b/./c").normalize());
System.out.println(Paths.get(".classpath").normalize());
System.out.println(Paths.get("/a/b/c/..").normalize());
System.out.println(Paths.get("../a/b/c").normalize());
```

  - This outputs:

```
/a/b/c
.classpath
/a/b
../a/b/c
```


Resolving/relativizing a Path

- **if you need to combine two paths, Resolve is to the rescue:**

```
Path dir = Paths.get("/home/java");
Path file = Paths.get("models/Model.pdf");
Path result = dir.resolve(file);
System.out.println("result = " + result);
```

This produces the absolute path by merging the two paths:

```
result = /home/java/models/Model.pdf
```

- **Relativizing a Path**
  - We have the absolute path of our home directory and the absolute path of the music file in our home directory. We want to know just the music file directory and name.

```
Path dir = Paths.get("/home/java");
Path music = Paths.get("/home/java/country/Swift.mp3");
Path mp3 = dir.relativize(music);
System.out.println(mp3);
```

  - This outputs :

```
country/Swift.mp3.
```

I/O vs. NIO.2 Permissions

Description	I/O Approach	NIO.2 Approach
Get the last modified date/time	`File file = new File("test");` `file.lastModified();`	`Path path = Paths.get("test");` `Files.getLastModifiedTime(path);`
Is read permission set	`File file = new File("test");` `file.canRead();`	`Path path = Paths.get("test");` `Files.isReadable(path);`
Is write permission set	`File file = new File("test");` `file.canWrite();`	`Path path = Paths.get("test");` `Files.isWritable(path);`
Is executable permission set	`File file = new File("test");` `file.canExecute();`	`Path path = Paths.get("test");` `Files.isExecutable(path);`

Set the last modified date/time (Note: `timeInMillis` is an appropriate `long`.)	`File file = new File("test");`
	`file.setLastModified(timeInMillis);`
		`Path path = Paths.get("test");`
		`FileTime fileTime = FileTime.`
		`fromMillis(timeInMillis);`
		`Files.setLastModifiedTime(path,`
		`fileTime);`

DirectoryStream

- **You might need to loop through a directory. Let's say you were asked to list out all the users with a home directory on this computer.**

```
/home
  | - users
        | - vafi
        | - eyra
```

```
Path dir = Paths.get("/home/users");
try (DirectoryStream<Path> stream =          // use try-with-resources
      Files.newDirectoryStream(dir)) {      // so we don't have close()
  for (Path path : stream)                  // loop through the stream
      System.out.println(path.getFileName());
}
```

  - As expected, this outputs
```
vafi
eyra
```

- **we only want the home directories of users whose names begin with either the letter *v* or the letter *w*.**

```
Path dir = Paths.get("/home/users");
try (DirectoryStream<Path> stream = Files.newDirectoryStream(
  dir, "[vw]*")) {                          // "v" or "w" followed by anything
for (Path path : stream)
      System.out.println(path.getFileName());
}
```

  - This outputs: vafi

Serialization

What is Serialization ?

- Serialization lets you simply say "save this object and all of its instance variables." unless It is explicitly marked a variable as transient
- Working with ObjectOutputStream and ObjectInputStream

```
ObjectOutputStream.writeObject()   // serialize and write

ObjectInputStream.readObject()     // read and deserialize
```

- Here's a small program that creates a Cat object, serializes it, and then deserializes it:

```
import java.io.*;

class Cat implements Serializable { }     // 1

public class SerializeCat {
  public static void main(String[] args) {
    Cat c = new Cat();                    // 2
    try {
      FileOutputStream fs = new FileOutputStream("testSer.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(c);                  // 3
      os.close();
    } catch (Exception e) { e.printStackTrace(); }

    try {
      FileInputStream fis = new FileInputStream("testSer.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      c = (Cat) ois.readObject();         // 4
      ois.close();
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Serializing Object Graphs

- You do have to make a conscious choice to create objects that are serializable by implementing the Serializable interface

```
import java.io.*;
public class SerializeDog {
  public static void main(String[] args) {
    Collar c = new Collar(3);
    Dog d = new Dog(c, 5);
    System.out.println("before: collar size is "
                       + d.getCollar().getCollarSize());
    try {
      FileOutputStream fs = new FileOutputStream("testSer.ser");
      ObjectOutputStream os = new ObjectOutputStream(fs);
      os.writeObject(d);
      os.close();
    } catch (Exception e) { e.printStackTrace(); }
    try {
      FileInputStream fis = new FileInputStream("testSer.ser");
      ObjectInputStream ois = new ObjectInputStream(fis);
      d = (Dog) ois.readObject();
      ois.close();
    } catch (Exception e) { e.printStackTrace(); }

    System.out.println("after:  collar size is "
                       + d.getCollar().getCollarSize());
  }
}
class Dog implements Serializable {
   private Collar theCollar;
   private int dogSize;
   public Dog(Collar collar, int size) {
     theCollar = collar;
     dogSize = size;
   }
   public Collar getCollar() { return theCollar; }
}
class Collar implements Serializable {
   private int collarSize;
   public Collar(int size) { collarSize = size; }
   public int getCollarSize() { return collarSize; }
}
```


How Inheritance Affects Serialization ?

- If a superclass is Serializable, then, according to normal Java interface rules, all subclasses of that class automatically implement Serializable implicitly.
- What happens if a superclass is not marked Serializable, but the subclass is?

```
class Animal { }
class Dog extends Animal implements Serializable {
   // the rest of the Dog code
}
```

- Getting back to our non-serializable Animal class with a serializable Dog subclass example:

```
class Animal {
   public String name;
}
class Dog extends Animal implements Serializable {
   // the rest of the Dog code
}
```

  - Because Animal is not serializable, **any state maintained in the Animal class, even though the state variable is inherited by the Dog, isn't going to be restored with the Dog when it's deserialized**! The reason is, the (unserialized) Animal part of the Dog is going to be reinitialized, just as it would be if you were making a new Dog (as opposed to deserializing one).

Serialization Is Not for Statics

- **Should static variables be saved as part of the object's state?**
  - Isn't the state of a static variable at the time an object was serialized important? Yes and no.
    - It might be important, but **it isn't part of the instance's state at all**.
- **Static variables are *never* saved as part of the object's state…because they do not belong to the object!**