Binary IO

CSC02A2

Outline

Outline

Binary IO

Binary IO

Binary IO

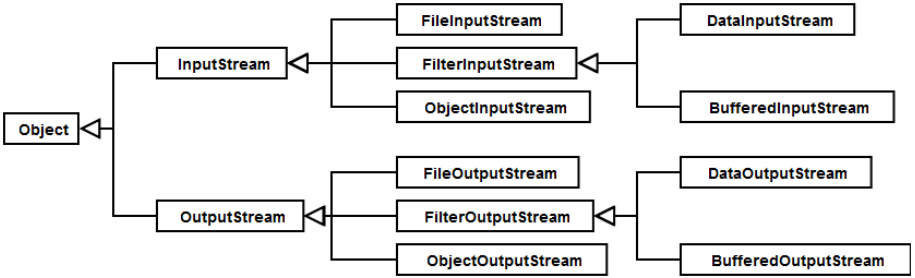
Binary IO is a means to read and write information to and from files stored on disk at a *byte* level.

- All information in modern computer systems is encoded as either the presence or absence of electric/magnetic charge (1s/0s). Therefore, there is no distinction between textual or binary files.
- Java (and most programming environments) provide text IO routines for automatic encoding/decoding[1] of textual data.
- Binary IO does not require this conversion and is therefore more efficient.
- Binary files produced by Java have agreed upon endianness[2] which makes them very portable.

[1] Encoding is a means of converting information into a required format (e.g. ASCII, Unicode, etc.) for storage or use. e.g. To write *199* to disk: The Unicode for *1* is 0x0031 which is then converted for storage based on the required encoding scheme of the file.

[2] Words can be stored in big-endian or little-endian format. Java uses big-endian where the most significant byte of a word is stored at a particular memory address with less significant bytes stored at higher addresses.

Binary IO Classes



FileOutputStream

```java
// Using a FileOutputStream to write bytes to a file.
FileOutputStream output = null;
try
{
  output = new FileOutputStream(new File("binfile.dat"));
  for (int k = 0; k < 10; k++)
  {
    output.write(k);
  }
}
catch (IOException ex)
{
  ex.printStackTrace();
}
finally
{
  if (output != null)
  {
    // Closing filestreams can result in an exception
    try
    {
      output.close();
    }
    catch (IOException ex)
    {
      ex.printStackTrace();
    }
  }
}
```


FileInputStream

```java
// Using a FileInputStream to read bytes from a file.
FileInputStream input = null;
try
{
  input = new FileInputStream(new File("binfile.dat");
  int value = -1; // -1 indicates that no more bytes to read
  while ((value = input.read()) != -1)
  {
    System.out.println(value);
  }
}
catch (IOException ex)
{
  ex.printStackTrace();
}
finally
{
  if (input != null)
  {
    // Closing filestreams can result in an exception
    try
    {
      input.close();
    }
    catch (IOException ex)
    {
      ex.printStackTrace();
    }
  }
}
```


Binary file contents

The resultant contents of the ***binfile.dat*** file (Not human-readable due to the encoding):

The resultant output when reading ***binfile.dat***:

```
0
1
2
3
4
5
6
7
8
9
```


Filtered IO

Filtered IO

Binary files operate at a **`byte`** level. Often an application will require to work with other types such as **primitive types**.

`DataInputStream`/**`DataOutputStream`** are concrete children of **`FilterInputStream`**/**`FilterOutputStream`** which implement filtering functionality for **primitive types**.

- The ***`available()`*** method returns a value for the number of bytes remaining in an input stream. This may be used to avoid an **`EOFException`**.
- To work with *primitive types* in a binary file, a ***filtered stream*** instance must be created. (**`DataInputStream`**/**`DataOutputStream`**)
- When working with a binary file, it is a good idea to make use of ***buffered stream***. This is due to the fact that disk access is an order of magnitude slower than accessing memory. (**`BufferedInputStream`**/**`BufferedOutputStream`**)

DataOutputStream

```java
// Work with filtered output stream
// Concrete class is DataOutputStream
DataOutputStream dataOut = null;
try
{
  // Create file output stream
  FileOutputStream fos = new FileOutputStream("bintypes.dat");
  // Buffer the file stream
  BufferedOutputStream bufOut = new BufferedOutputStream(fos);
  // Finally filtered stream
  dataOut = new DataOutputStream(bufOut);
  // Write primitive types
  dataOut.writeUTF("Name");
  dataOut.writeDouble(123.456);
}
catch (IOException ex)
{
  ex.printStackTrace();
}
finally
{
  if (dataOut != null)
  {
    try
    {
      dataOut.close();
    }
    catch (IOException ex)
    {
      ex.printStackTrace();
    }
  }
}
```


DataInputStream

```java
// Work with filtered input stream
// Concrete class is DataInputStream
DataInputStream dataIn = null;
try
{
  // Create file input stream
  FileInputStream fin = new FileInputStream("bintypes.dat");
  // Buffer the file stream
  BufferedInputStream bufIn = new BufferedInputStream(fin);
  // Finally filtered stream
  dataIn = new DataInputStream(bufIn);
  // Read primitive types
  String name = dataIn.readUTF();
  double value = dataIn.readDouble();
  System.out.format("%s - %f", name, value);
}
catch (IOException ex)
{
  ex.printStackTrace();
}
finally
{
  if (dataIn != null)
  {
    try
    {
      dataIn.close();
    }
    catch (IOException ex)
    {
      ex.printStackTrace();
    }
  }
}
```


Object IO

Object IO

More complex applications require more than just **`byte`**s and ***primative types***. Complex applications usually work with objects in some way.

The **`ObjectInputStream`**/**`ObjbectOutputStream`** classes handle the (de)serialization of objects. These classes can also handle ***primitive data*** that makes them especially flexible.

Writing an object to a binary file requires ***`serializing`*** it. Serialization converts an object into a sequence of bytes to be processed. To allow serialization, objects only need implement the **`java.io.Serializable`** ***marker*** interface. For an *array* to be serialized, all of its elements must be serializable.

When writing object to disk, there are things which *cannot* be serialized.

- ***`static`*** data cannot be serialized.
- ***`transient`*** data members are also not included in serialization.

When any object is serialized, it is given a unique ***`serialID`*** that prevents it from being streamed to to disk twice.

Serializable Class

```java
// Class must implement the marker interface Serializable
public class Storeable implements java.io.Serializable
{
  // Static variables are not serialized
  private static double    STATIC_DBL = 3.14;
  // Variables modified to be transient are not serialized
  private transient String  testName;
  // This array will be serialized
  private int[]        state;

  public Storeable(int[] state)
  {
    this.testName="A";
    this.state = new int[state.length];
    System.arraycopy(state, 0, this.state, 0, state.length);
  }
}
```


ObjectOutputStream

```java
// Work with object output stream
// Concrete class is DataInputStream
ObjectOutputStream objOut = null;
try
{
  // Create file output stream
  FileOutputStream fos = new FileOutputStream("bintypes.dat");
  // Buffer the file stream
  BufferedOutputStream bufOut = new BufferedOutputStream(fos);
  // Finally filtered stream
  objOut = new ObjectOutputStream(bufOut);
  // Write object types
  int[] ary = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  Storeable a = new Storeable(ary);
  objOut.writeObject(a);
}
catch (IOException ex)
{
  ex.printStackTrace();
}
finally
{
  if (objOut != null)
  {
    try
    {
      objOut.close();
    }
    catch (IOException ex)
    {
      ex.printStackTrace();
    }
  }
}
```


ObjectInputStream

```java
// Work with object input stream
ObjectInputStream objIn = null;
try
{
  // Create file input stream
  FileInputStream fin = new FileInputStream("bintypes.dat");
  // Buffer the file stream
  BufferedInputStream bufIn = new BufferedInputStream(fin);
  // Finally object stream
  objIn = new ObjectInputStream(bufIn);
  // Read object types
  Object objFromFile =  in.readObject();
  if(objFromFile instanceof Storeable)
  {
    Storeable b = (Storeable) objFromFile;
    // Process object as needed
  }
}
catch (IOException ex)
{
  ex.printStackTrace();
}
finally
{
  if (objIn != null)
  {
    try
    {
      objIn.close();
    }
    catch (IOException ex)
    {
      ex.printStackTrace();
    }
  }
}
```


Random Access Files

Random Access Files

All of the streams that have been examined so far have been either *read-only* or *write-only* and have pointed to sequential files. An alternate approach must be used for file that require read and write access at the same time.

The **RandomAccessFile** class allows for read and write actions to be performed without closing the file and opening it with different access (if opened in read/write mode).

- ***seek(n)*** method will move to **byte** *n* in the file.
- ***getFilePointer*** method will retrieve the current position in the file.
- ***read()***/***write()*** methods read/write integers and byte arrays.
- ***length()*** method returns the length of the file.

RandomAccessFiles also contain filtered input/output methods such as ***writeDouble()*** as well.