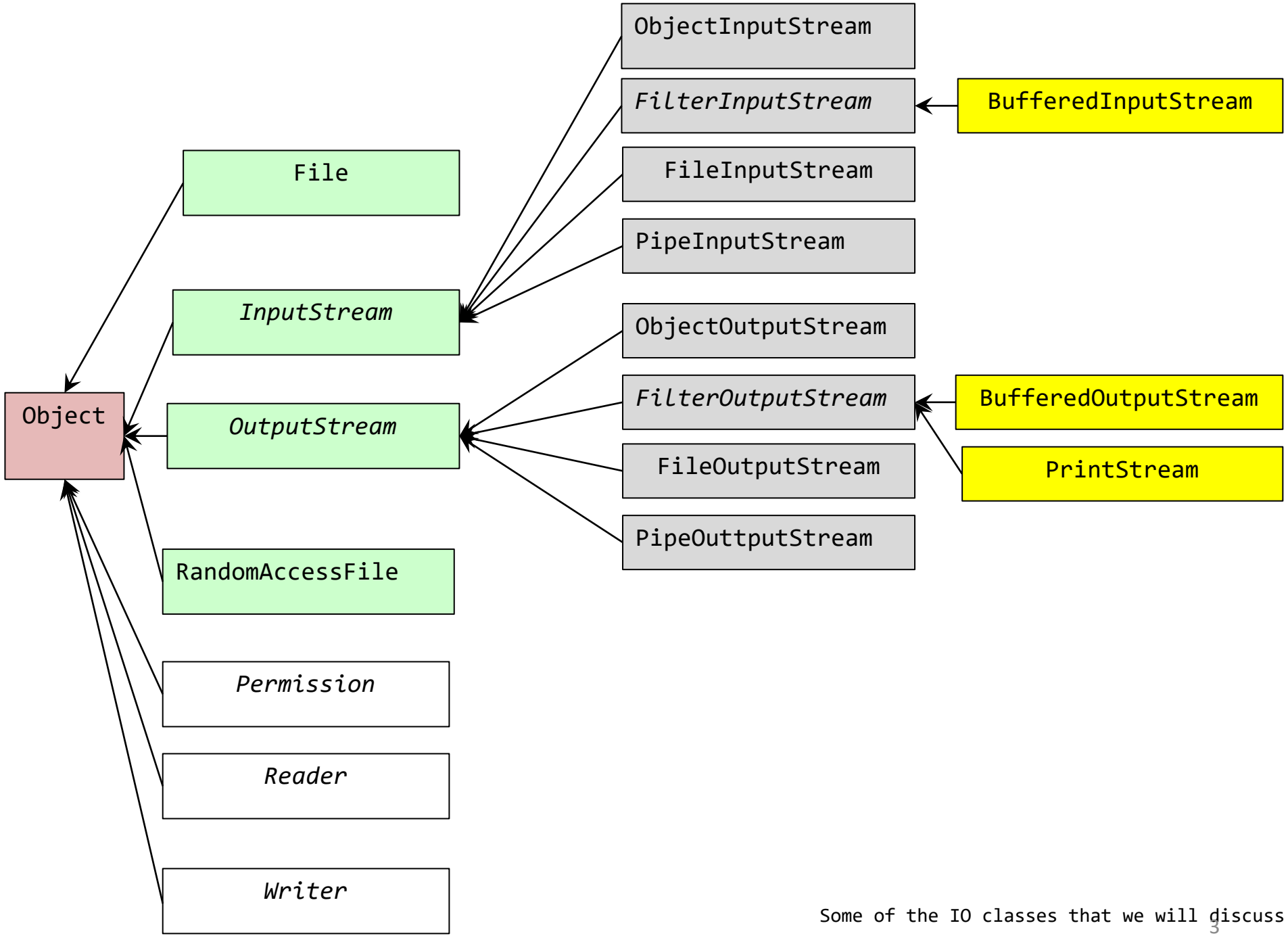
ENSF 607

6 - Java Object Streams (Part I)

Java I/O Streams

- Java views each file as a sequential stream of bytes
- Operating system (OS) provides the mechanism:
  - To determine end of file
  - Count of total bytes in file
- Java program processing a stream of bytes receives an indication from the OS when program reaches **end of stream**



Some of the IO classes that we will discuss

Java Streams

- **Byte-based streams** – read and write data as stream of 8-bit bytes.
  - Classes that read or write **bytes**, derived from two abstract classes:
    - `InputStream`
    - `OutputStream`
- **Character-based streams** – read and write data as a sequence of characters
  - Classes that read and write character, derived from two abstract classes:
    - `Reader`
    - `Writer`

Java Streams (2)

- **Classes that allow random access**. Useful for direct access applications such as transaction-processing, airline-reservation system, and point of sale systems.
- **Class File** which is useful for retrieving information about a file or a directory on the disk…

Byte-Based Streams

Byte-Based Stream (1)

- Classes derived from **InputStream** and **OutputStrearm**:
  - Classes **FileInputStream** and **FileOutputStream**, used to read from and write files.
  - Classes **PipeInputStream** and **PipeOutputStream**, used to establish data channel between threads. One thread sends data to another thread by writing to a PipeOutputStream. And the target thread reads the data from pipe using PipeInputStream.

Byte-Based Stream (2)

- Abstract classes `FilterInputStream` and `FilterOutputStream`, provide additional functionality such as buffering:
  - Class `PrintStream` a subclass of `FilterOutputStream`, performs text output to a specific stream. `System.out` and `System.err` are objects of this stream.
  - One of the added features of `PrintStream` is that the `flush` method is automatically invoked after:
    - a byte array is written,
    - `println` methods is invoked,
    - or a newline character or byte ('\n') is written.

Byte Based Stream Example :

```
public class MyClass {
  FileInputStream in = null;
  FileOutputStream out = null;

  public void foo() {
     try {
      in = new FileInputStream("in.txt");
      out = new FileOutputStream("out.txt");
      int c;

      while ((c = in.read()) != -1) {
         out.write(c);
         System.out.print(c);
      }
      } catch (FileNotFoundException e) {
      // DO SOMETHING
      } catch (IOException e1) {
         e1.printStackTrace();
      }
```

```
  finally {
    if (in != null) {
       try {
         in.close();
       } catch (IOException e) {
         e.printStackTrace();
       }
    }
    if (out != null) {
       try {
         out.close();
       }catch (IOException e) {
         e.printStackTrace();
       }
    }
   } // End of finally
} // end of class MyClass
```

Character Based Streams

Character-Based Streams

- Character stream classes are descendants of two abstract classes:
  - **Reader**
  - **Writer**
- As with byte streams, there are character stream classes that specialize in file I/O:
  - **FileReader**
  - **FileWriter**

Character Based Stream Example

```
public class MyClass {

   FileReader inputStream = null;
   FileWriter outputStream = null;
   public void foo() throws IOException {
         try {
                  inputStream = new FileReader("in.txt");
                  outputStream = new FileWriter("out.txt");
                  int c;
                  while ((c = inputStream.read()) != -1) {
                           outputStream.write(c);
                  }
                  }catch (IOException e){/*Do something*/}
                  finally {
                  if (inputStream != null) {
                           inputStream.close();
                  }
                  if (outputStream != null) {
                           outputStream.close();
                  }
            }
    }
}
```

Buffered Streams

Buffered Streams

- **Unbuffered I/O stream** is much **less efficient**, since it often triggers disk access, network activity, or some other operation expensive operations.
- **Buffered input streams** read data from or write data from ***buffer***.
- An **unbuffered stream** can be converted into a buffered stream, by wrapping the unbuffered stream within the constructor of a buffered stream class. Example:

```
inputStream = new BufferedReader(new FileReader("input.txt"));
outputStream = new BufferedWriter(new FileWriter("output.txt"));
```

- There are **four buffered stream classes**:
  - **BufferedInputStream (byte base)**
  - **BufferedOutputStream (byte base)**
  - **BufferedReader (character base)**
  - **BufferedWriter (character base)**

Flushing Buffered Streams

- To empty the buffer at certain points, without waiting for it to fill, is known as **flushing the buffer**.
- Some buffered output class constructors have an optional argument for **autoflush**.
  - autoflush must be true to be on.
  - The autoflush causes the buffer to be flushed at certain events (such as call to println or format methods).

- Or, you can **flush a stream manually**, by invoking the `flush` method.
  - The flush method is available to any output stream, but has no effect unless the stream is buffered.

Object Serialization

Object Serialization

- **Object serialization** – mechanism to read or write an entire object from a file
- **Serialized object** – object represented as sequence of bytes, includes object's data and type information about object
- **Deserialization** – recreate object in memory from data in file
- Serialization and deserialization performed with classes `ObjectInputStream` and `ObjectOutputStream`, methods `readObject` and `writeObject`

Object Serialization:

- Programmers must declare a class to implement the Serializable interface
- To open a file, create a FileOutputStream wrapped by an ObjectOutputStream
  - ObjectOutputStream method writeObject writes object to output file.
  - ObjectOutputStream method close closes both objects

Reading and Deserializing Data

- To open a file for reading objects, create a FileInputStream wrapped by an ObjectInputStream
  - FileInputStream provides methods for reading byte-based input from a file
  - Use ObjectInputStream method readObject to reads an object form a file.
    - EOFException occurs if attempt made to read past end of file
    - ClassNotFoundException occurs if the class for the object being read cannot be located
  - ObjectInputStream method close closes both objects.

Open File to Deserialize Objects

```
public class MyClass {

  private ObjectInputStream input;

  public void foo() throws IOException {
      try {
          input = new ObjectInputStream(new FileInputStream("clients.ser"));
      } catch (IOException ioException) {
      System.err.println("Error opening file.");
  }
}
```

Reading Objects

```
try // input the values from the file
{
    while ( true )
    {
        record = ( Account ) input.readObject();
        // display record contents on the screen
         System.out.printf( "%-10d%-12s%-12s%10.2f\n",
                           record.getAccount(), record.getFirstName(),
                           record.getLastName(), record.getBalance() );
  }
}
catch ( EOFException e)
{
   System.err.println("Error ….");
}
```

Example

```
import java.io.Serializable;

public class Account implements Serializable {

   private int account;
   private String firstName;
   private String lastName;
   private double balance;

  public Account ()
  {
    this( 0, "", "", 0.0 );
  }

  public Account ( int acc, String first, String last, double bal)
  {
    setAccount( acc );
    setFirstName( first );
    setLastName( last );
  }
}
```

Open file for writing Objects

```
private ObjectOutputStream output;

try
{
  output = new ObjectOutputStream( new FileOutputStream( "records.ser" ) );
}
catch ( IOException ioException )
{
   System.err.println( "Error opening file." );
}
```

Writing Objects to the File

```
Account record;
try
{
     record = new Account( accountNumber, firstName, lastName, balance );
     output.writeObject( record );
}
catch ( IOException ioException )
{
    System.err.println( "Error writing to file." );
} // end catch
catch ( NoSuchElementException elementException )
{
    System.err.println( "Invalid input. Please try again." );
} // end catch
```

Closing Files of Objects

```
try // close file
{
    if (output != null)
       output.close();
} // end try
catch(IOException ioException)
{
   System.err.println("Error closing file.");
   System.exit(1);
} // end catch
```

SerialVersionUID

- The serialization runtime, associates a **version number**, called a serialVersionUID with each serializable class.
  - It is used to verify that the sender and receiver have loaded classes for that object that are compatible with respect to serialization.
  - If the receiver has loaded a class for the object that has a different `serialVersionUID` than that of the corresponding sender's class, then deserialization will result in an `InvalidClassException`.

SerialVersionUID cont'd

- A serializable class can declare its own `serialVersionUID` explicitly by declaring a field named `"serialVersionUID"` that must be `static`, `final`, and of type `long`.
- **Example:**

```
static final long serialVersionUID = 42L;
```

  - If a serializable class does not explicitly declare a `serialVersionUID`, then the serialization runtime will calculate a default `serialVersionUID` value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification.
  - It is ***strongly recommended*** that all serializable classes explicitly declare `serialVersionUID`.