PROGRAMMING III
OOP. JAVA LANGUAGE

COURSE 10

PREVIOUS COURSE CONTENT

- **Graphical User Interfaces**
  - Abstract Window Toolkit
  - Swing

COUSE CONTENT

- **Input/Output Sreams**
  - Text Files
  - Byte Files
  - RandomAcessFile
- **Exceptions**
- **Serialization**
- **NIO**

WHAT IS A FILE?

WHAT IS A FILE?

- A file is a collection of data in mass storage.
- The same file can be read or modified by different programs.
- The program must be aware of the format of the data in the file.
- The files are maintained by the operating system.
- The system provides commands and/or GUI utilities for viewing file directories and for copying, moving, renaming, and deleting files.
- The operating system also provides basic functions, callable from programs, for reading and writing directories and files.

FILE TYPES

- **Text files**
  - A computer user distinguishes text (“ASCII”) files and “binary” files. This distinction is based on how you treat the file.
  - A text file is assumed to contain lines of text (for example, in ASCII code).
  - Each line terminates with a newline character (or a combination, carriage return plus line feed).
  - Examples:
    - Any plain-text file, typically named something.txt
    - Source code of programs in any language (for example, Something.java)
    - HTML documents
    - .....

FILE TYPES

Binary Files

- A “binary” file can contain any information, any combination of bytes.
- Only a programmer / designer knows how to interpret it.
- Different programs may interpret the same file differently (for example, one program displays an image, another extracts an encrypted message).
- Examples:
  - Compiled programs (for example, Something.class)
  - Image files (for example, something.gif)
  - Music files (for example, something.mp3)
- Any file can be treated as a binary file (even a text file, if we forget about the special meaning of CR-LF).

STREAM

- **Stream**
  - A stream is a connection to a source of data or to a destination for data (sometimes both)
  - An input stream may be associated with the keyboard
  - An input stream or an output stream may be associated with a file
  - Different streams have different characteristics:
    - A file has a definite length, and therefore an end
    - Keyboard input has no specific end

STREAM

- A stream is an abstraction derived from sequential input or output devices.
- An input stream produces a stream of characters; an output stream receives a stream of characters, “one at a time.”
- Streams apply not just to files, but also to IO devices, Internet streams, and so on.
- A file can be treated as an input or output stream.
- In reality file streams are buffered for efficiency: it is not practical to read or write one character at a time from or to mass storage.
- It is common to treat text files as streams.

FILES AND STREAMS

- **Java views each files as a sequential stream of bytes**

- **Operating system provides mechanism to determine end of file**
  - End-of-file marker
  - Count of total bytes in file
  - Java program processing a stream of bytes receives an indication from the operating system when program reaches end of stream

FILES AND STREAMS

- **File streams**
  - Byte-based streams – stores data in binary format
    - Binary files – created from byte-based streams, read by a program that converts data to human-readable format
  - Character-based streams – stores data as a sequence of characters
    - Text files – created from character-based streams, can be read by text editors

- **Java opens file by creating an object and associating a stream with it**

- **Standard streams – each stream can be redirected**
  - `System.in` – standard input stream object, can be redirected with method `setIn`
  - `System.out` – standard output stream object, can be redirected with method `setOut`
  - `System.err` – standard error stream object, can be redirected with method `setErr`

I/O API

- **I/O (input/outpu)**
  - refers to the interface between a computer and the rest of the world
  - between a single program and the rest of the computer

- **java.io.***
  - Stream oriented
  - Blocking IO
- **java.nio.* (java version ≥ 1.7)**
  - Buffer oriented
  - Non blocking IO
  - Selectors

IO API

BufferedInputStream
BufferedOutputStream
BufferedReader
BufferedWriter
ByteArrayInputStream
ByteArrayOutputStream
CharArrayReader
CharArrayWriter
DataInputStream
DataOutputStream
File
FileDescriptor
FileInputStream
FileOutputStream
FilePermission
FileReader
FileWriter
FilterInputStream
FilterOutputStream
FilterReader
FilterWriter
InputStream
InputStreamReader
LineNumberInputStream
LineNumberReader
ObjectInputStream
ObjectInputStream.GetField
ObjectOutputStream
ObjectOutputStream.PutField
ObjectStreamClass
ObjectStreamField
OutputStream
OutputStreamWriter
PipedInputStream
PipedOutputStream
PipedReader
PipedWriter
PrintStream
PrintWriter
PushbackInputStream
PushbackReader
RandomAccessFile
Reader
SequenceInputStream
SerializablePermission
StreamTokenizer
StringBufferInputStream
StringReader
StringWriter
Writer

IO API

- **Uses four hierarchies of classes**
  - Reader
  - Writer
  - InputStream
  - OutputStream.
- **InputStream/OutputStream hierarchies deal with bytes. Reader/Writer hierarchies deal with chars.**
- **Has a special stand-alone class RandomAccessFile.**
- **The Scanner class has been added to java.util in Java 5 to facilitate reading numbers and words.**

IO. USAGE

- **IO flow**
  - import java.io.*;
  - *Open* the stream
    - There is data external to your program that you want to get, or you want to put data somewhere outside your program
    - When you open a stream, you are making a connection to that external place
    - Once the connection is made, you forget about the external place and just use the stream
  - *Use* the stream (read, write, or both)
    - Using a stream means doing input from it or output to it
    - But it's not usually that simple--you need to manipulate the data in some way as it comes in or goes out
  - *Close* the stream
    - A stream is an expensive resource
    - There is a limit on the number of streams that you can have open at one time
    - You should not have more than one stream open on the same file
    - You must close a stream before you can open it again
    - *Always close your streams*

JAVA.IO.FILE

- **The File class represents a file (or folder) in the file directory system.**
- **Class `File` useful for retrieving information about files and directories from disk**
- **Objects of class `File` do not open files or provide any file-processing capabilities**

```
String pathname = "../Data/words.txt“;
File file = new File(pathname);
```

- **Methods:**
  - String getName( ) - returns file name
  - boolean exists() - returns true if the file exists
  - String getAbsolutePath( ) - return the absolute file path
  - long length( ) - return the size of file
  - boolean isDirectory( ) - return true if the file is a directory
  - File[ ] list( ) - returns the list of the directory

JAVA.IO.FILE

- **Class `File` provides four constructors:**
  - Takes `String` specifying name and path (location of file on disk)
  - Takes two `String`s, first specifying path and second specifying name of file
  - Takes `File` object specifying path and `String` specifying name of file
  - Takes `URI` object specifying name and location of file

- **Different kinds of paths**
  - Absolute path
    - contains all directories, starting with the root directory, that lead to a specific file or directory
  - Relative path
    - normally starts from the directory in which the application began executing

JAVA.IO.FILE

- **Separator character – used to separate directories and files in a path**
  - Windows uses \
  - UNIX uses /
  - Java process both characters, `File.pathSeparator` can be used to obtain the local computer's proper separator character

- **Common Programming Error**
  - Using \ as a directory separator rather than \\ in a string literal is a logic error. A single \ indicates that the \ followed by the next character represents an escape sequence. Use \\ to insert a \ in a string literal.

IO. READING FROM STANDARD INPUT

- **Can use**
  - BufferedReader
    - BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in))
    - How to read?
      - int read()
        - returns character code, reads one charcter
      - String readLine()
        - returns a line of text
      - ...
  - Scanner
    - Scanner stdin = new Scanner(System.in)
    - How to read?
      - int nextInt()
      - double nextDouble()
      - String nextLine()
      - ...

IO. READING FROM TEXT FILES

- **Can use**
  - BufferedReader
  - Scanner
  - LineNumberReader
    - String readLine()
      - reads a line from a file
    - int getLineNumber()
      - returns the number of lines read from the file so far

IO. READING FROM TEXT FILES

```
public class ReadingFromFile {
    public static void main(String[] args) throws IOException {
        // opening the file for reading
        FileReader f = new FileReader("test.txt");

        // creation of the object for reading
        BufferedReader in = new BufferedReader(f);

        // reading a line of text from the file
        String line = in.readLine();
        System.out.println(line);

        // closing the file
        f.close();
    }
}
```

IO STREAMTOKENIZER

- **Parses inputStreams into "tokens", allowing the tokens to be read one at a time**
- **Can recognize identifiers, numbers, quoted strings, and various comment styles.**
- **Example**

```
public class StreamTokenizerDemo {
  public static void main(String[] args) {
        try {
      // create an ObjectInputStream for the file we created before
      ObjectInputStream ois =  new ObjectInputStream(new FileInputStream("test.txt"));

      // create a new tokenizer
      Reader r = new BufferedReader(new InputStreamReader(ois));   StreamTokenizer st = new StreamTokenizer(r);

      // print the stream tokens
      boolean eof = false;
      do {
        int token = st.nextToken();
        switch (token) {
          case StreamTokenizer.TT_EOF:  System.out.println("End of File encountered."); eof = true;   break;
          case StreamTokenizer.TT_EOL:   System.out.println("End of Line encountered.");   break;
          case StreamTokenizer.TT_WORD: System.out.println("Word: " + st.sval);   break;
          case StreamTokenizer.TT_NUMBER:      System.out.println("Number: " + st.nval);    break;
          default:  System.out.println((char) token + " encountered.");    if (token == '!') {   eof = true;  }
        }
      } while (!eof);
    } catch (Exception ex) {        ex.printStackTrace();      } } }
```

IO. WRITING TO TEXT FILES

- **Can Use**
  - PrintWriter
    - void print()
    - PrintWriter printf()
    - void println()
  - Example

```
public static void main(String[] args) throws IOException {
…
PrintWriter outFile = new PrintWriter(“results.txt”);
outFile.println(“ANALYSIS for “ + infileName);
outFile.print(“Number of samples”);
…
outFile.close();
```

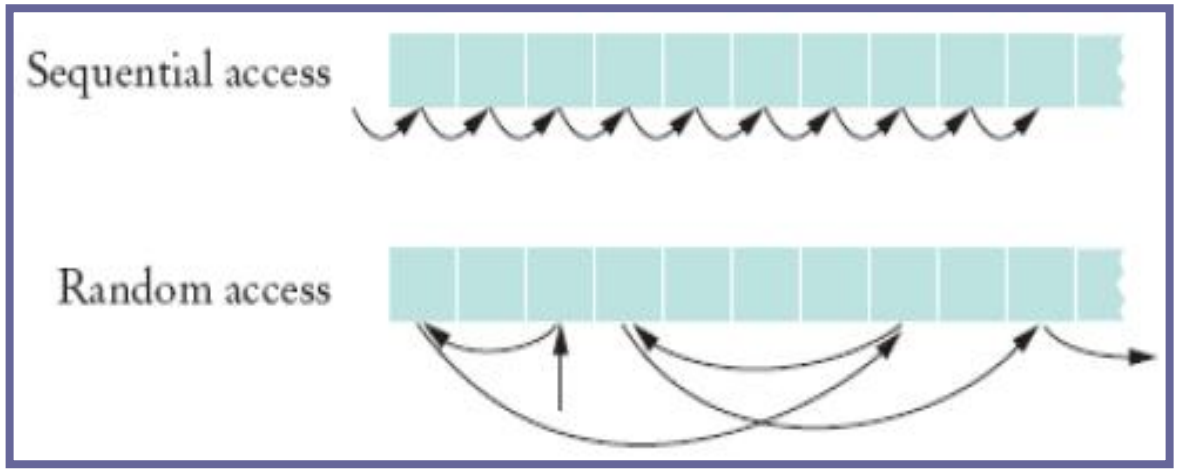
IO READING/WRITING BYTES

- **To read and write 8-bit bytes, programs should use the byte streams, descendants of InputStream and OutputStream .**
- **InputStream and OutputStream provide the API and partial implementation for *input streams* (streams that read 8-bit bytes) and *output* streams (streams that write 8-bit bytes).**
- **These streams are typically used to read and write binary data such as images and sounds.**
- **Example**

```
private static void copyFileUsingFileStreams(File source, File dest)  throws IOException {
        InputStream input = null;
        OutputStream output = null;
        try {
                input = new FileInputStream(source);
                output = new FileOutputStream(dest);
                byte[] buf = new byte[1024];
                int bytesRead;
                while ((bytesRead = input.read(buf)) > 0)  output.write(buf, 0, bytesRead);
        } finally {
                input.close();
                output.close();
        }
}
```

IO RANDOM ACESS FILES

- **Random access files are files in which records can be accessed in any order**
  - Also called direct access files
  - More efficient than sequential access files



IO RANDOM ACESS FILES

- **NOT compatible with the stream/reader/writer models**
- **With a random-access file, you can seek to the desired position and then read and write an amount of bytes**
- **Only support seeking relative to the beginning of the file**
  - Not relative to current position of file pointer
  - However there are methods that report the current position

IO RANDOM ACESS FILES

- **Methods**
  - long getFilePointer()
    Returns the current offset in this file.
  - long length()
    Returns the length of this file.
  - void seek(long pos)
    Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.

IO RANDOM ACESS FILES

- RandomAccessFile(File file, String mode)
  - Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
- RandomAccessFile(String name, String mode)
  - Creates a random access file stream to read from, and optionally to write to, a file with the specified name.
- **The mode should be either “r” , “rw”, “rws” or “rwd”**
  - rws
    - flushes the contents of the file and the modification date of the file.
  - rwd
    - flushs the contents of the file, but the modification date might not change until the file is closed.
  - rw
    - only flushes when you tell it to and doesn't change the modifcation date until you close the file.

IO RANDOM ACESS FILES

- **Constructors**
  - When a RandomAccessFile is created in read-only mode a FileNotFoundException is generated
  - When a RandomAccessFile is created in read-write a zero length file will be created

IO RANDOM ACESS FILES

- **File pointers**
  - RandomAccessFile supports *file pointer* which indicates the current location in the file.
  - When the file is first created, the file pointer is set to 0, indicating the beginning of the file.
  - Calls to the read and write methods adjust the file pointer by the number of bytes read or written.

IO RANDOM ACESS FILES

- **Manipulate file pointers**
  - RandomAccessFile contains three methods for explicitly manipulating the file pointer.
    - int skipBytes(int) — Moves the file pointer forward the specified number of bytes
    - void seek(long) — Positions the file pointer just before the specified byte
    - long getFilePointer() — Returns the current byte location of the file pointer
  - Usage
    - To move the file pointer to a specific byte
      ```
      f.seek(n);
      ```
    - To get current position of the file pointer.
      ```
      long n = f.getFilePointer();
      ```
    - To find the number of bytes in a file
      ```
      long filelength = f.length();
      ```

IO RANDOM ACESS FILES. EXAMPLE

```
public class RandomAccess {
    public static void main(String args[]) throws IOException {
        RandomAccessFile myfile = new RandomAccessFile("rand.dat",
"rw");
        myfile.writeInt(120);
        myfile.writeDouble(375.50);
        myfile.writeInt('A'+1);
        myfile.writeBoolean(true);
        myfile.writeChar('X');

        // set pointer to the beginning of file and read next two
items
        myfile.seek(0);
        System.out.println(myfile.readInt());
        System.out.println (myfile.readDouble());

        //set pointer to the 4th item and read it
          myfile.seek(16);
          System.out.println(myfile.readBoolean());
```

IO RANDOM ACESS FILES. EXAMPLE

```
        // Go to the end and "append" an integer 2003
        myfile.seek(myfile.length());
        myfile.writeInt(2003);
        // read 5th and 6th items
        myfile.seek(17);
        System.out.printl(myfile.readChar());
        System.out.println(myfile.readInt());
        System.out.println("File length: "+myfile.length());
        myfile.close();
    }
}
```

IO EXCEPTIONS

- FileNotFoundException
- IOException

SERIALIZATION

- **Persistence**
  - Saving information about an object to recreate at different time, or place or both.

- **Object serialization**
  - implementing persistence: convert object's state into byte stream to be used later to reconstruct (build-deserialized) a virtually identical copy of original object.

- **Default serialization for an object writes**
  - the class of the object,
  - the class signature,
  - values of all non-transient and non-static fields.

SERALIAZTION

- **Classes for serialization**
  - For serialization:
    - `java.io.ObjectOutputStream` via `writeObject` which calls on `defaultWriteObject`,
  - For deserialization:
    - `java.io.ObjectInputStream` via `readObject` which calls on `defaultReadObject`.
- **Any object instance that belongs to the graph of the object being serialized must be serializable as well.**
- **Superclass must be Serializable.**
  - This interface is an empty interface and is used to mark the objects of such class as persistent.

SERIALIZATION

- **Serialization**
  - It writes the values of a class members
- **Deserialization**
  - It reads values written during serialization
  - Static fields in the class are left untouched.
    - If class needs to be loaded, then normal initialization of the class takes place, giving static fields its initial values.
  - Transient fields will be initialized to default values
  - Recreation of the object graph will occur in reverse order from its serialization.

SERIALIZATION

- **Conditions for serializability**
  - If an object is to be serialized
    - The class must be declared as public
    - The class must implement Serializable
    - The class must have a no-argument constructor
    - All fields of the class must be serializable: either primitive types or serializable objects

SERIALIZATION

- **To Write into an ObjectOutputStream**
  - FileOutputStream out = new FileOutputStream(“afile”) ;
    ObjectOutputStream oos = new ObjectOutputStream(out) ;
    oos.writeObject(“Today”) ;
    oos.writeObject(new Date()) ;
    oos.flush() ;

- **To Read from an ObjectInputStream**
  - FileInputStream in = new FileInputStream(“afile”) ;
    ObjectInputStream ois = new ObjectInputStream(in) ;
    String today = (String) ois.readObject() ;
    Date date = (Date) ois.readObject() ;

SERIALIZATION

- **Custom Serialization**
  - Create own complete serialization by implementing the interface **Externalizable**

```
interface Externalizable{
    public void writeExternal(ObjectOutput out)
      throws IOException;
    public void readExternal(ObjectInput in)
      throws IOException;
}
```

  - writeExternal and readExternal must save/load the state of the object. They must explicitly coordinate with its supertype to save its state.

SERIALIZABLE VS. NON-SERIALIZABLE OBJECTS

- Java.lang.Object does not implement serializable, so you must decide which of your classes need to implement it.
- AWT, Swing components, strings, arrays are defined serializable.
- Certain classes and subclasses are not serializable: Thread, OutputStream, Socket
- When a serializable class contains instance variables which are not or should not be serializable they should be marked as that with the keyword `transient`.

SERIALIZATION. TRANSIENT FIELDS

- **These fields will not be serialized.**

- **When deserialized, these fields will be initialized to default values**
  - Null for object references
  - Zero for numeric primitives
  - False for boolean fields

- **If these values are unacceptable**
  - Provide a readObject() that invokes defaultReadObject() and then restores transient fields to their acceptable values.
  - Or, the fields can be initialized when used for the first time. (Lazy initialization.)

SERIALIZATION. SERIAL VERSION UID

- **You should explicitly declare a serial version UID in every serializable class.**
  - Eliminates serial version UID as a potential source of incompatibility.
  - Small performance benefit, as Java does not have to come up with this unique number.
  - private static final long serialVersionUID =rlv;
    - rlv can be any number out thin air, but must be unique for each serializable class in your development.
  - If you want to make a new version of the class incompatible with existing version, choose a different UID. Deserialization of previous version will fail with InvalidClassException.

SERIALZATION. PERFORMANCE

- **Serialization is a very expensive process.**
  - You must clearly have reasons to serialize instead of you directly writing what you need to save about the state of an object.

SERIALZATION

- **Default or Customized serialization?**
  - Allowing a class's instances to be serializable can be as simple as adding the words "implements Serializable" to the class specification.
  - This is a common misconception, the truth is far more complex.
  - While efficiency it is one cost associated with it, there are other long-term costs that are much more substantial.
  - Using default serialization is very easy but this a very specious

SERIALIZATION

- **Costs**
  - A major cost is that it decreases flexibility to change a class's implementation once the class has been release
  - Increases the likelihood of bugs and security holes.
  - Increases the testing associated with releasing a new version of the class.
  - Classes design for inheritance should rarely implement serializable and interfaces should rarely extend it.
    - You should provide parameterless constructor on non-serializable classes designed for inheritance, in case it is subclassed and the subclass wants to provide serialization.
  - Inner classes should rarely if ever, implement Serializable.
  - A static member class can be serializable.

NIO

- **Four key new helper Types new in Java 7**
- **Class java.nio.file.Paths**
  - Exclusively static methods to return a Path by converting a string or Uniform Resource Identifier (URI)
- **Interface java.nio.file.Path**
  - Used for objects that represent the location of a file in a file system, typically system dependent
- **Class java.nio.file.Files**
  - Exclusively static methods to operate on files, directories and other types of files
- **Class java.nio.file.FileSystem**
- **Typical use case:**
  - Use Paths to get a Path. Use Files to do stuff.

NIO

- **Way NIO?**
  - Methods didn’t throw exceptions when failing
  - Rename worked inconsistently
  - No symbolic link support
  - Additional support for meta data
  - Inefficient file meta data access
  - File methods didn’t scale
  - Walking a tree with symbolic links not possible

NIO

- **File copy is really easy**
  - With fine grain control

```
Path src = Paths.get("/home/fred/readme.txt");
Path dst = Paths.get("/home/fred/copy_readme.txt");

Files.copy(src, dst,
    StandardCopyOption.COPY_ATTRIBUTES,
    StandardCopyOption.REPLACE_EXISTING);
```

- **File move is supported**
  - Optional atomic move supported

```
Path src = Paths.get("/home/fred/readme.txt");
Path dst = Paths.get("/home/fred/copy_readme.txt");

Files.copy(src, dst,
   StandardCopyOption.COPY_ATTRIBUTES,
   StandardCopyOption.REPLACE_EXISTING);
```

NIO

- **Files helper class is feature rich:**
  - Copy
  - Create Directories
  - Create Files
  - Create Links
  - Use of system “temp” directory
  - Delete
  - Attributes – Modified/Owner/Permissions/Size, etc.
  - Read/Write

NIO

- **DirectoryStream iterate over entries**
  - Scales to large directories
  - Uses less resources
  - Smooth out response time for remote file systems
  - Implements `Iterable` and `Closeable` for productivity
- **Filtering support**
  - Build-in support for glob, regex and custom f

```
Path srcPath = Paths.get("/home/fred/src");

try (DirectoryStream<Path> dir =
   srcPath.newDirectoryStream("*.java")) {
for (Path file : dir)
   System.out.println(file.getName());
}
```

NIO

- Path and Files are “link aware”
- createSymbolicLink(Path, Path, FileAttribute<?>)

```
Path newLink = Paths.get(. . .);
Path existingFile = Paths.get(. . .);
try {
    Files.createSymbolicLink(newLink, existingFile);
} catch (IOException x) {
    System.err.println(x);
} catch (UnsupportedOperationException x) {
     //Some file systems or some configurations
     //may not support links
    System.err.println(x);
}
```

NIO

- A FileVisitor interface makes walking a file tree for search, or performing actions, trivial.
- SimpleFileVisitor implements

```
preVisitDirectory(T dir, BasicFileAttributes attrs);
visitFile(T dir, BasicFileAttributes attrs);
visitFileFailed(T dir, IOException exc);
postVisitDirectory(T dir, IOException exc);
```

```
SAMPLE:
    Path startingDir = ...;
    PrintFiles pf = new PrintFiles(); // SimpleFileVisitor sub
         // visitFile(Path p, BasicFileAttributes bfa) {
      //        System.out.println(file.getFileName());}
    Files.walkFileTree(startingDir, pf);
```

NIO

- **Watching a Directory**
  - Create a WatchService “watcher” for the filesystem
  - Register a directory with the watcher
  - “Watcher” can be polled or waited on for events
    - Events raised in the form of Keys
    - Retrieve the Key from the Watcher
    - Key has filename and events within it for create/delete/modify
  - Ability to detect event overflows

NIO

- **Custom FileSystems**
  - FileSystems class is factory to great FileSystem (interface)
  - Java 7 allows for developing custom FileSystems, for example:
    - Memory based or zip file based systems
    - Fault tolerant distributed file systems
    - Replacing or supplementing the default file system provider
  - Two steps:
    - Implement `java.nio.file.spi.FileSystemProvider`
      - URI, Caching, File Handling, etc.
    - Implement `java.nio.file.FileSystem`
      - Roots, RW access, file store, etc.