Core Java

Java IO framework

- Input/Output functionality in Java is provided under package java.io and java.nio package.
- IO framework is used for File IO, Network IO, Memory IO, and more.
- File is a collection of data and information on a storage device.
- File = Data + Metadata
- Two types of APIs are available file handling
    - FileSystem API -- Accessing/Manipulating Metadata
    - File IO API -- Accessing/Manipulating Contents/Data

java.io.File class

- A path (of file or directory) in file system is represented by "File" object.
- Used to access/manipulate metadata of the file/directory.
- Provides FileSystem APIs
    - String[] list() -- return contents of the directory
    - File[] listFiles() -- return contents of the directory
    - boolean exists() -- check if given path exists
    - boolean mkdir() -- create directory
    - boolean mkdirs() -- create directories (child + parents)
    - boolean createNewFile() -- create empty file
    - boolean delete() -- delete file/directory
    - boolean renameTo(File dest) -- rename file/directory
    - String getAbsolutePath() -- returns full path (drive:/folder/folder/...)
    - String getPath() -- return path
    - File getParentFile() -- returns parent directory of the file
    - String getParent() -- returns parent directory path of the file
    - String getName() -- return name of the file/directory
    - static File[] listRoots() -- returns all drives in the systems.
    - long getTotalSpace() -- returns total space of current drive
    - long getFreeSpace() -- returns free space of current drive
    - long getUsableSpace() -- returns usable space of current drive
    - boolean isDirectory() -- return true if it is a directory
    - boolean isFile() -- return true if it is a file
    - boolean isHidden() -- return true if the file is hidden
    - boolean canExecute()
    - boolean canRead()
    - boolean canWrite()
    - boolean setExecutable(boolean executable) -- make the file executable
    - boolean setReadable(boolean readable) -- make the file readable
    - boolean setWritable(boolean writable) -- make the file writable
    - long length() -- return size of the file in bytes
    - long lastModified() -- last modified time

    - boolean setLastModified(long time) -- change last modified time

Java IO

- Java File IO is done with Java IO streams.
- Stream is abstraction of data source/sink.
    - Data source -- InputStream or Reader
    - Data sink -- OutputStream or Writer
- Java supports two types of IO streams.
    - Byte streams (binary files) -- byte by byte read/write
    - Character streams (text files) -- char by char read/write
- All these streams are AutoCloseable (so can be used with try-with-resource construct)

IO Framework

Chaining IO Streams

- Each IO stream object performs a specific task.
    - FileOutputStream -- Write the given bytes into the file (on disk).
    - BufferedOutputStream -- Hold multiple elements in a temporary buffer before flushing it to underlying stream/device. Improves performance.
    - DataOutputStream -- Convert primitive types into sequence of bytes. Inherited from DataOutput interface.
    - ObjectOutputStream -- Convert object into sequence of bytes. Inherited from ObjectOutput interface.
    - PrintStream -- Convert given input into formatted output.
    - Note that input streams does the counterpart of OutputStream class hierarchy.
- Streams can be chained to fulfil application requirements.

Primitive types IO

- DataInputStream & DataOutputStream -- convert primitive types from/to bytes
    - primitive type --> DataOutputStream --> bytes --> FileOutputStream --> file.
        - DataOutput interface provides methods for conversion - writeInt(), writeUTF(), writeDouble(), ...
    - primitive type <-- DataInputStream <-- bytes <-- FileInputStream <-- file.
        - DataInput interface provides methods for conversion - readInt(), readUTF(), readDouble(), ...

DataOutput/DataInput interface

- interface DataOutput
    - writeUTF(String s)
    - writeInt(int i)
    - writeDouble(double d)
    - writeShort(short s)
    - ...
- interface DataInput

    - String readUTF()
    - int readInt()
    - double readDouble()
    - short readShort()
    - ...

Serialization

- ObjectInputStream & ObjectOutputStream -- convert java object from/to bytes
    - Java object --> ObjectOutputStream --> bytes --> FileOutputStream --> file.
        - ObjectOutput interface provides method for conversion - writeObject().
    - Java object <-- ObjectInputStream <-- bytes <-- FileInputStream <-- file.
        - ObjectInput interface provides methods for conversion - readObject().
- Converting state of object into a sequence of bytes is referred as Serialization. The sequence of bytes includes object data as well as metadata.
- Serialized data can be further saved into a file (using FileOutputStream) or sent over the network (Marshalling process).
- Converting (serialized) bytes back to the Java object is referred as Deserialization.
- These bytes may be received from the file (using FileInputStream) or from the network (Unmarshalling process).

ObjectOutput/ObjectInput interface

- interface ObjectOutput extends DataOutput
    - writeObject(obj)
- interface ObjectInput extends DataInput
    - obj = readObject()

Serializable interface

- Object can be serialized only if class is inherited from Serializable interface; otherwise writeObject() throws NotSerializableException.
- Serializable is a marker interface.

transient fields

- writeObject() serialize all non-static fields of the class. If fields are objects, then they are also serialized.
- If any field is intended not to serialize, then it should be marked as "transient".
- The transient and static fields (except serialVersionUID) are not serialized.

serialVersionUID field

- Each serializable class is associated with a version number, called a serialVersionUID.
- It is recommended that programmer should define it as a static final long field (with any access specifier). Any change in class fields expected to modify this serialVersionUID.

```
private static final long serialVersionUID = 1001L;
```

- During deserialization, this number is verified by the runtime to check if right version of the class is loaded in the JVM. If this number mismatched, then InvalidClassException will be thrown.
- If a serializable class does not explicitly declare a serialVersionUID, then the runtime will calculate a default serialVersionUID value for that class (based on various aspects of the class described in the Java(TM) Object Serialization specification).

Buffered streams

- Each write() operation on FileOutputStream will cause data to be written on disk (by OS). Accessing disk frequently will reduce overall application performance. Similar performance problems may occur during network data transfer.
- BufferedOutputStream classes hold data into a in-memory buffer before transferring it to the underlying stream. This will result in better performance.
    - Java object --> ObjectOutputStream --> BufferedOutputStream --> FileOutputStream --> file on disk.
- Data is sent to underlying stream when buffer is full or flush() called explicitly.
- BufferedInputStream provides a buffering while reading the file.
- The buffer size can be provided while creating the respective objects.

Chaining IO Streams

- Each IO stream object performs a specific task.
    - FileOutputStream -- Write the given bytes into the file (on disk).
    - BufferedOutputStream -- Hold multiple elements in a temporary buffer before flushing it to underlying stream/device. Improves performance.
    - DataOutputStream -- Convert primitive types into sequence of bytes. Inherited from DataOutput interface.
    - ObjectOutputStream -- Convert object into sequence of bytes. Inherited from ObjectOutput interface.
    - PrintStream -- Convert given input into formatted output.
    - Note that input streams does the counterpart of OutputStream class hierarchy.
- Streams can be chained to fulfil application requirements.

Primitive types IO

- DataInputStream & DataOutputStream -- convert primitive types from/to bytes
    - primitive type --> DataOutputStream --> bytes --> FileOutputStream --> file.
        - DataOutput interface provides methods for conversion - writeInt(), writeUTF(), writeDouble(), ...
    - primitive type <-- DataInputStream <-- bytes <-- FileInputStream <-- file.
        - DataInput interface provides methods for conversion - readInt(), readUTF(), readDouble(), ...

DataOutput/DataInput interface

- interface DataOutput
    - writeUTF(String s)
    - writeInt(int i)
    - writeDouble(double d)
    - writeShort(short s)
    - ...
- interface DataInput
    - String readUTF()
    - int readInt()
    - double readDouble()
    - short readShort()
    - ...

Serialization

- ObjectInputStream & ObjectOutputStream -- convert java object from/to bytes
    - Java object --> ObjectOutputStream --> bytes --> FileOutputStream --> file.
        - ObjectOutput interface provides method for conversion - writeObject().
    - Java object <-- ObjectInputStream <-- bytes <-- FileInputStream <-- file.
        - ObjectInput interface provides methods for conversion - readObject().
- Converting state of object into a sequence of bytes is referred as Serialization. The sequence of bytes includes object data as well as metadata.
- Serialized data can be further saved into a file (using FileOutputStream) or sent over the network (Marshalling process).
- Converting (serialized) bytes back to the Java object is referred as Deserialization.
- These bytes may be received from the file (using FileInputStream) or from the network (Unmarshalling process).

ObjectOutput/ObjectInput interface

- interface ObjectOutput extends DataOutput
    - writeObject(obj)
- interface ObjectInput extends DataInput
    - obj = readObject()

Serializable interface

- Object can be serialized only if class is inherited from Serializable interface; otherwise writeObject() throws NotSerializableException.
- Serializable is a marker interface.

transient fields

- writeObject() serialize all non-static fields of the class. If fields are objects, then they are also serialized.
- If any field is intended not to serialize, then it should be marked as "transient".

- The transient and static fields (except serialVersionUID) are not serialized.

serialVersionUID field

- Each serializable class is associated with a version number, called a serialVersionUID.
- It is recommended that programmer should define it as a static final long field (with any access specifier). Any change in class fields expected to modify this serialVersionUID.

```
private static final long serialVersionUID = 1001L;
```

- During deserialization, this number is verified by the runtime to check if right version of the class is loaded in the JVM. If this number mismatched, then InvalidClassException will be thrown.
- If a serializable class does not explicitly declare a serialVersionUID, then the runtime will calculate a default serialVersionUID value for that class (based on various aspects of the class described in the Java(TM) Object Serialization specification).

Buffered streams

- Each write() operation on FileOutputStream will cause data to be written on disk (by OS). Accessing disk frequently will reduce overall application performance. Similar performance problems may occur during network data transfer.
- BufferedOutputStream classes hold data into a in-memory buffer before transferring it to the underlying stream. This will result in better performance.
    - Java object --> ObjectOutputStream --> BufferedOutputStream --> FileOutputStream --> file on disk.
- Data is sent to underlying stream when buffer is full or flush() called explicitly.
- BufferedInputStream provides a buffering while reading the file.
- The buffer size can be provided while creating the respective objects.

PrintStream class

- Produce formatted output (in bytes) and send to underlying stream.
- Formatted output is done using methods print(), println(), and printf().
- System.out and System.err are objects of PrintStream class.

Scanner class

- Added in Java 5 to get the formatted input.
- It is java.util package (not part of java io framework).

```
Scanner sc = new Scanner(inputStream);
// OR
Scanner sc = new Scanner(inputFile);
```

- Helpful to read text files line by line.

Character streams

- Character streams are used to interact with text file.
- Java char takes 2 bytes (unicode), however char stored in disk file may take 1 or more bytes depending on char encoding.
    - https://www.w3.org/International/questions/qa-what-is-encoding
- The character stream does conversion from java char to byte representation and vice-versa (as per char encoding).
- The abstract base classes for the character streams are the Reader and Writer class.
- Writer class -- write operation
    - void close() -- close the stream
    - void flush() -- writes data (in memory) to underlying stream/device.
    - void write(char[] b) -- writes char array to underlying stream/device.
    - void write(int b) -- writes a char to underlying stream/device.
- Writer Sub-classes
    - FileWriter, OutputStreamWriter, PrintWriter, BufferedWriter, etc.
- Reader class -- read operation
    - void close() -- close the stream
    - int read(char[] b) -- reads char array from underlying stream/device
    - int read() -- reads a char from the underlying device/stream. Returns -1
- Reader Sub-classes
    - FileReader, InputStreamReader, BufferedReader, etc.

Process vs Threads

Program

- Program is set of instructions given to the computer.
- Executable file is a program.
- Executable file contains text, data, rodata, symbol table, exe header.

Process

- Process is program in execution.
- Program (executable file) is loaded in RAM (from disk) for execution. Also OS keep information required for execution of the program in a struct called PCB (Process Control Block).
- Process contains text, data, rodata, stack, and heap section.

Thread

- Threads are used to do multiple tasks concurrently within a single process.
- Thread is a lightweight process.
- When a new thread is created, a new TCB is created along with a new stack. Remaining sections are shared with parent process.

Process vs Thread

- Process is a container that holds resources required for execution and thread is unit of execution/scheduling.

- Each process have one thread created by default -- called as main thread.

Process creation (Java)

- In Java, process can be created using Runtime object.
- Runtime object holds information of current runtime environment that includes number of processors, JVM memory usage, etc.
- Current runtime can be accessed using static getRuntime() method.

```
Runtime rt = Runtime.getRuntime();
```

- The process is created using exec() method, which returns the Process object. This object represents the OS process and its waitFor() method wait for the process termination (and returns exit status).

```
String[] args = { "/path/of/executable", "cmd-line arg1", ... };
Process p = rt.exec(args);
int exitStatus = p.waitFor();
```

Multi-threading (Java)

- Java applications are always multi-threaded.
- When any java application is executed, JVM creates (at least) two threads.
  - main thread -- executes the application main()
  - GC thread -- does garbage collection (release unreferenced objects)
- Programmer may create additional threads, if required.

Thread creation

- To create a thread
  - step 1: Implement a thread function (task to be done by the thread)
  - step 2: Create a thread (with above function)
- Method 1: extends Thread

```
class MyThread extends Thread {
    @Override
    public void run() {
        // task to be done by the thread
    }
}
```

```
MyThread th = new MyThread();
th.start();
```

- Method 2: implements Runnable

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        // task to be done by the thread
    }
}
```

```
MyRunnable runnable = new MyRunnable();
Thread th = new Thread(runnable);
th.start();
```

- Java doesn't support multiple inheritance. If your class is already inherited from a super class, you cannot extend it from Thread class. Prefer Runnable in this case; otherwise you may choose any method.

```
// In Java GUI application is inherited from Frame class.
// to create run() in the same class, you must use Runnable
class MyGuiApplication extends Frame implements Runnable {
    // ...
    public void run() {
        // ...
    }
    // ...
}
```

start() vs run()

- run():
  - Programmer implemented code to be executed by the thread.
- start():
  - Pre-defined method in Thread class.
  - When called, the thread object is submitted to the (JVM/OS) scheduler. Then scheduler select the thread for execution and thread executes its run() method.

Thread methods

- static Thread currentThread()
  - Returns a reference to the currently executing thread object.
- static void sleep(long millis)

    - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

- static void yield()
    - A hint to the scheduler that the current thread is willing to yield its current use of a processor.
- Thread.State getState()
    - Returns the state of this thread.
    - State can be NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
- void run()
    - If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called. If thread class extends from Thread class, this method should be overridden. The default implementation is empty.
- void start()
    - Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
- void join()
    - Waits for this thread to die/complete.
- boolean isAlive()
    - Tests if this thread is alive.
- void setDaemon(boolean daemon);
    - Marks this thread as either a daemon thread (true) or a user thread (false).
- boolean isDaemon()
    - Tests if this thread is a daemon thread.
- long getId()
    - Returns the identifier of this Thread.
- void setName(String name)
    - Changes the name of this thread to be equal to the argument name.
- String getName()
    - Returns this thread's name.
- void setPriority(int newPriority)
    - Changes the priority of this thread.
    - In Java thread priority can be 1 to 10.

    - May use predefined constants MIN_PRIORITY(1), NORM_PRIORITY(5), MAX_PRIORITY(10).

- int getPriority()
    - Returns this thread's priority.
- ThreadGroup getThreadGroup()
    - Returns the thread group to which this thread belongs.
- void interrupt()
    - Interrupts this thread -- will raise InterruptedException in the thread.
- boolean isInterrupted()
    - Tests whether this thread has been interrupted.

Daemon threads

- By default all threads are non-daemon threads (including main thread).
- We can make a thread as daemon by calling its setDaemon(true) method -- before starting the thread.
- Daemon threads are also called as background threads and they support/help the non-daemon threads.
- When all non-daemon threads are terminated, the Daemon threads get automatically terminated.