Head First Java: Chapter 15 Notes

September 16, 2019

Network and Connection in Java

In the server-client model of networking, all clients (that is, individual users) that desired to be connected to the same network make a connection to the same *server*. Data and information is not send directly from client to client, it is send from client to server, then server to the desired receiving client. In Java, connections are made between `Socket` objects, which represents a network connection between two entities (such as a client and a server.) There are many types of `Socket` objects in Java, in terms of connection made via the Internet, new `Socket` objects can be instantiated with the destination's Internet Protocol address (IP address) and the Transmission Control Port (TCP) number: `Socket socket = new Socket("196.164.1.103", 5000)`, for example (the `Socket` constructor can be passed different configurations of arguments.) The IP address represents the "location" of the connection destination on the Internet, and the TCP number represents the type of application the connection is for (*e.g.* Email (POP3 or SMTP etc.), websites (HTTP or HTTPS), file transfer (SSH or FTP)). Almost all network operations in Java require importing the `Java.net.*` package. Reading data from and writing data to the network is similar to serializing data in Java:

- Reading data from the network:
  1. Instantiate a `Socket` object with the IP address and TCP number of the location where the incoming data is coming from.
  2. Instantiate an `InputStreamReader` object by passing an `InputStream` object with the data from the network. An `InputStream` can be instantiated from the `Socket` object by calling its `getInputStream()` method.
  3. Instantiate a `Bufferedreader` object by passing its constructor the `InputStreamReader`. Data then can be read by called the reader's methods, such as `readLine()` for text.
- Writing data to be send to the network:
  1. Instantiate a `Socket` object with the IP address and TCP number of the location where the incoming data is coming from.
  2. Instantiate a `PrintWriter` object by passing its constructor the `OutputStreamReader` object from the `Socket` object by calling its `getOutputStream()` method.

3. Write data by calling methods from the `PrintWriter` object, such as the `print()` and `println()` methods for text.

The above steps are for client-side connections. To program a server in Java, the process is similar but with some slight modification:

1. A program acting as a server should instantiate a `ServerSocket` object instead. The constructor of `ServerSocket` only needs the TCP number.

2. A normal `Socket` object is then instantiated by passing the `accept()` method of the `ServerSocket` (and then assigning it to a object reference variable as usual.) From this `Socket` reading and writing data is the same as previously described.

Threading in Java

Like most other programming language, Java can only handle one task at a time, unless multi-threading is utilize. In object-oriented Java, this is implemented with the `Thread` class object and its methods, available in the standard library. Multi-threading in Java can visualizing as having multiple memory stack, where each has its independent stack of method/constructor frames, with frames the top frame in scope and popping off once the method concludes its task. However, it is important to keep in mind that unless the computer where the multi-threading occurs truly utilizes multiple processor and the JVM accounted for it, then multi-threading is not truly simultaneously operating method frames from multiple stacks (because a single-core CPU in a computer can truly only handle one operation at a time, albeit extremely quickly.) Rather, it switches between the multiple stacks rapidly as the task of the frames are being completed, so that it *appears* that there is true multi-threading. How the stack-switching occurs (how quickly the switches are, which stack goes first, what order, etc.) is completely dependent on the JVM and generally out of control from the user.

To implement multi-threading in Java:

1. Any methods from a class that needs to be run on a separate stack from the *main* stack (that is, the stack where the `main()` method is running on), that class must implement the `Runnable` interface. The `Runnable` interface has a `run()` method that must be overridden if implemented. Essentially, all methods to be ran on this new thread/stack (apart form the `main()` stack) should be "bundled" into this `Runnable` class object.

2. Instantiate a `Thread` object, with the `Runnable` object (of whatever class) as argument.

3. Call the `start()` method of the `Thread` object instance.

Below is an example execution of this process:

```
public class MyRunnable implements Runnable {

    public void doMore() {
        System.out.println(''top of the stack'');
    }

    public void go() {
        doMore();
    }

    public void run() {
        go();
    }
}

class ThreadTester {

    public static void main (String[] args) {
        Runnable threadJob = new MyRunnable();
        Thread myThread = new Thread(threadJob);

        myThread.start();
        System.println(''back in main'')
    }
}
```

The above code will in effect create two stacks, the `main()` stack, then the `myThread.start()` stack. As discussed previously, once the second (the non-`main()`) stack is created via the `start()` method, method frames on both stack will run seeming simultaneously. The JVM contains a thread scheduler that is generally not controllable by the user that switches between the threads. While a thread is not in scope (because the JVM thread scheduler decided to run another stack/thread) that thread is said to be at a *runnable* state. The thread scheduler can also move a thread into a *blocked* state, where it is temporarily non-runnable. There are several reasons why the JVM might do this because of conflicting threads (*e.g.* waiting for data to be available) or it was coded to be blocked using the `sleep()` method. Because the unpredictability of the JVM thread scheduler, tasks operating by different threads may not operate in the intended order. However, some control can be achieved if on each thread `sleep()` is called for a few millisecond. As a static method, `sleep()` does not need an instance of `Thread` object to be called, and will temporarily

put a thread to the blocked state for however long the user specified (passed as argument.) This in effect will enable the JVM thread scheduler to pick another thread to be in scope, akin to "giving a chance" for the other thread to run.

Concurrency Issues

When utilizing more than two threads in multi-threading, the user may run into concurrency issues. Two separate thread might need access to the same data, but one thread could have modify data without the other anticipating it, and thus causing errors. As such, it is necessary to *lock* certain process until they are completed to prevent confusion. Declaring methods with the `synchronized` keyword, this modify the method so that only one thread may access it at anytime. Note that if a class contains multiple `synchronized` methods, all of them as a group can only be accessed by one thread at a time. That is, if class A has `synchronized` methods `alpha()` and `beta()`, if `alpha()` is being accessed by thread `one`, it would mean that both `alpha()` and `beta()` are inaccessible by other threads until `one` has completed its task with `alpha()`. It is important to note that while synchronizing is essential to prevent concurrency issues, abusing synchronization can lead to performance drop and deadlock. Hence, synchronizing methods should only be done at a necessity basis.