Java Overview Part II

Based on Notes by J. Johns

(based on Java in a Nutshell, Learning Java)

Also Java Tutorial, Concurrent Programming in Java

Access Control

- **Public** – everyone has access
- **Private** – no one outside this class has access
- **Protected** – subclasses have access
- **Default** – package-access

Final Modifier

- final class – cannot be subclassed
- final method – cannot be overriden
- final field – cannot have its value changed. Static final fields are compile time constants.
- final variable – cannot have its value changed

Static Modifier

- static method – a class method that can only be accessed through the class name, and does not have an implicit *this* reference.
- static field – A field that can only be accessed through the class name. There is only 1 field no matter how many instances of the class there are.

Topics

- Concurrent Programming in Java
  - Threads
  - Synchronization
  - Groups/Priorities
  - Inter-thread communication
- java.io package
  - Streams
  - Readers/Writers
  - Object serialization/persistence

Concurrent Programming

- What is a Thread?
- What can go wrong with a Thread?
- How can we fix it?
- How does Java handle them?
- Why would I use them?

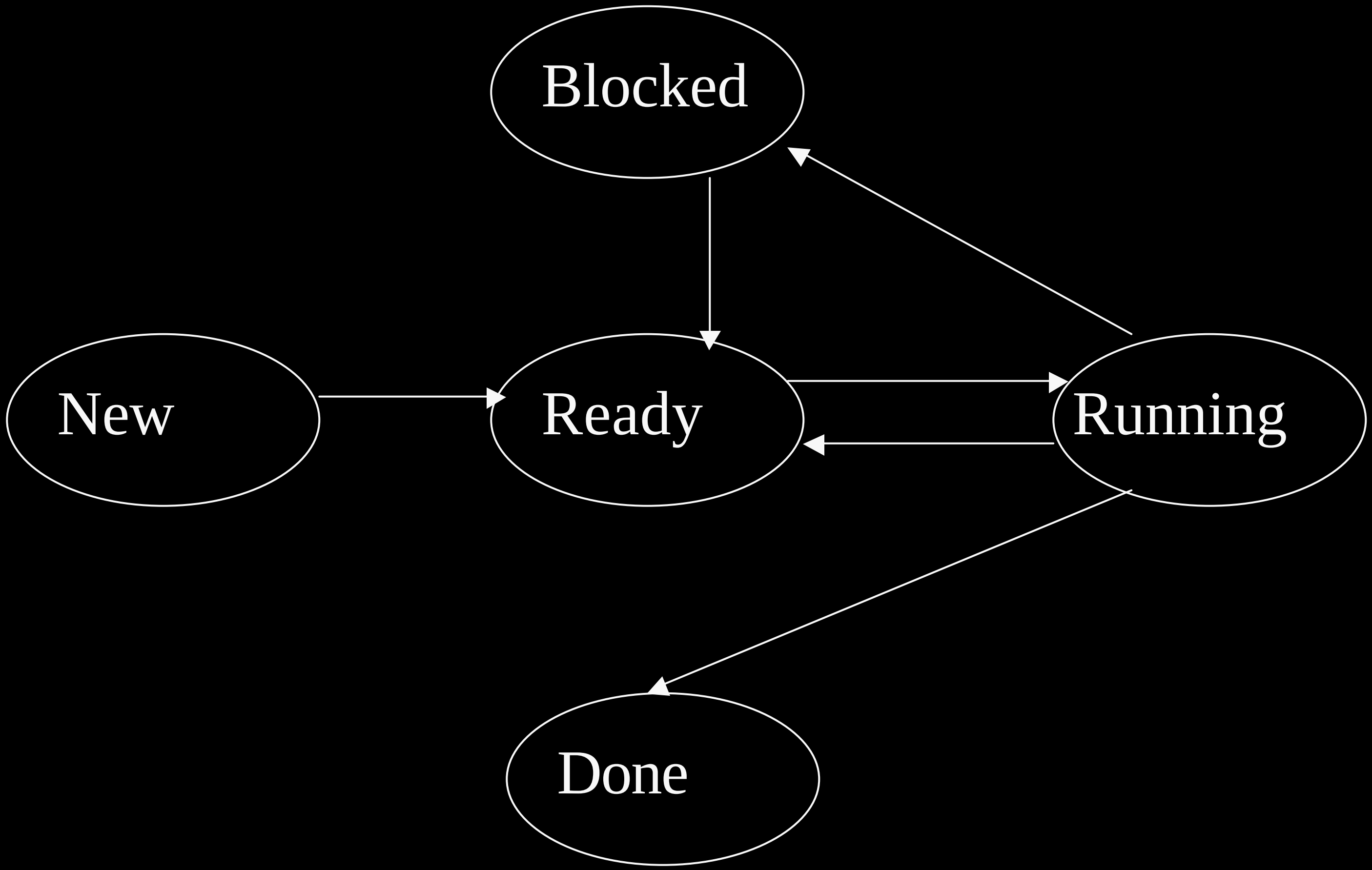
What is a Thread?

- A Thread is a *single, sequential flow of control within a program*.
- Now they are so ubiquitous that you don’t notice any more.
  - DOS vs. Windows

Processes vs. Threads

- Processes
  - Completely separate, unrelated concurrent execution on the level of the operating system. (eg multiple programs running at the same time)
- Threads
  - Concurrent units of execution within a given program. (eg pulling down a menu while loading a web page within a web browser)

Life cycle of a Thread



Life cycle of a Thread (cont’d)

- The OS can interrupt the thread at **any** time while it is running, and allow **any** other thread to run.
- Threads can put themselves into a wait state until another thread wakes them up.

What can go wrong?

- Assuming that threads, existing in the same program, have access to the same variable.
- What if one is reading data from an array, gets interrupted, and another one writes to that array, even though the thread wanted the old values?

What can you do about it?

- `synchronized` is a keyword that can be applied to a method to say that one and only one thread will have access to this method at a time.

```
public synchronized void blah()
  { }
```

More on `synchronized`

- `synchronized` deals with locks on a given object. Every object only has 1 lock. It can be used as a statement inside a block of code instead of on a whole method.

```
{ …
  synchronized (o) { … }
}
```

More on synchronized

```
public synchronized void blah()
 {…}
```

Is the same as

```
public void blah () {
  synchronized (this) {
  }
}
```

More that can go wrong

- What happens if you have two things that do this - deadlock

```
public void doSomething() {
Synchronized (a) {
  Synchronized (b) {
  // code
  } }
}
```

```
public void doOther() {
Synchronized (b) {
  Synchronized (a) {
  // code
  } }
}
```

How does Java handle Threads?

- Subclass java.lang.Thread, or implement java.lang.Runnable.
- After you instantiate a thread, it is in the *ready* state.
- To start running a Thread, call the start method on it.

How does Java handle them? (cont’d)

- To implement Runnable, you need to have a method that override

  ```
  public void run();
  ```

- This is the method that implements the running section of the thread life-cycle.
- The thread dies (stops) when the run method terminates.

How does Java handle them? (cont’d)

- The run method may be interrupted at any time by the operating system and put into the interrupted state, but that’s not something you really need to handle.

How does Java handle them? (cont’d)

- Run methods generally look like:

```
public class SomeThread extends
  Thread {
    public void run() {
        while (notDone) {…}
    }
    public void finish() {
        notDone = false;
    }
}
```

Thread examples

- Test: Ten threads printing their name three times.
- Test2: Main thread joins printing threads.
- Test3: Each thread yields after printing.
- Test4: Printing threads yield randomly.

Advanced Thread Features

- All Java Threads have a priority. If you want a thread to run more relative to other threads, give it a higher priority.
- Threads can be grouped in ThreadGroup objects
- Test7: Two groups of threads
- Test8: Printing threads with priority

Why would I use them?

- Most advance programs rely on Threads for various tasks.
- ThreadLister Example
- 2 cases:
  - When you want to be doing 2 different things simultaneously.
  - When you have a large problem that can be broken up and solved in smaller sections, or large I/O bound processes.

Inter-Thread Communication

- Sometimes one thread may be interested in the activities of another. Or, one could have a functional dependency on another.
  - Reading from a file or over a network?
  - Waiting for a given thread to return a result.
  - Polling (Busy Waiting) vs. Notification
  - BadConsumer Example

Waiting for notification

- As defined in object, every object has a wait(), notify(), and notifyAll() method.
  - These should never be overridden
- They can only be called from inside `synchronized` blocks, and they only effect other threads in synchronized blocks which are synchronized on the same object.

wait() (cont’d)

- When a thread enters a *wait* state, it does nothing until it is *notified* by another thread.
- It also gives up it’s lock on the object when wait is called.

```
public synchronized blah() {
  wait();
  … // do something
}
```

notify()

- To awaken a thread, a different thread which has a lock on the same object must call notify.
- When notify is called, the block that had the lock on the object continues to have it's lock it releases it.
  - **Then** a thread is awakened from its wait() and can grab the lock and continue processing.

notify() (cont’d)

- Note that you don’t specify what is being awoken in notify(). If there are more than 1 thread waiting on the same condition, you have no control of which awakens.
- notify() only awakens 1 thread.
- notifyAll() awakens **all** threads.

notifyAll() (cont’d)

- There are two versions - notify() and notifyAll().
- Notify is safe only under 2 conditions:
  - When only 1 thread is waiting, and thus guaranteed to be awakened.
  - When multiple threads are waiting on the same condition, and it doesn’t matter which one awakens.
- In general, use notifyAll()

Break?

- Is it time for a break?
- I hope so.

java.io Abstract Classes

- There are 4 abstract classes for java.io that are very analogous to one another, but they do slightly different things.
  - Reader
  - Writer
  - InputStream
  - OutputStream

Reader/Writer

- Used to read and write text.
- They are very nice because they handle unicode character conversion for you.
- Methods provided:
  - int read()
  - int read(char[] buf)
  - int read(char[] buf, int offset, int len)

InputStream/OutputStream

- Used to read and write everything else
- Methods provided:
  - int read()
  - int read(byte[] buff)
  - int read(byte[] buff, int offset, int len)
- In general, for every read method, there is a write method.

File Input and Output

- Something useful !!!
- FileReader/Writer and FileInputStream/OutputStream
- In general use the readers and writers for text files, and streams for when you don’t know what may be in the files.
- Example

cat.java notes

- Separate Exception checking
- read() returns -1 when at the end of the file.
- Reading and writing are ***always*** done inside try/catch statements.
  - Why?
- I used the 0-parameter read, but ones with arrays work faster.

cat2.java

- BufferedReader is nice for its readLine() method.
- Note: *Wrapping* streams/readers is not quite like calling a copy constructor.
  - Since any type of reader/stream can be buffered, after a stream is created with a certain source, it is passed to a constructor for use of that class's special properties.

Wrapping Streams/Readers

- Streams/Readers extend their parent classes with more complex ways to read/write data than the 3 basic read/write methods
- BufferedReader is a wrapper that allows lines of data to be read at a time. Internally, it’s just using the 3 basic read/write methods.

Wrapping (cont’d)

- What’s the big deal?
- By wrapping streams, functionality can be added easily to any I/O operation to and destination.

Specialized Wrappers

- Buffering can be done for every stream.
- You can filter any I/O stream to provide a layer between input and output.
- Data of different primitive types can be read/written with DataInputStreams.
- PrintStream is used for terminal type textual representation of data.
- There are I/O wrappers for arrays and Strings as well.

More specialized wrappers

- There are pipes to allow communication between threads
- There are additional packages java.util.zip and java.util.jar to read and write zip and jar files, using the same I/O paradigm.
- Gzip example

What to notice from gzip.java

- Streams can be wrapped several levels.
- References are declared outside the try/catch block, and instantiated inside.
- We're still using basic read/write methods, but they work well with buffers.
- All streams should **ALWAYS** be closed separately.

More complex exception handling

- Finally clause - code in it is always run, irregardless of if an exception is thrown or not.
  - Any code you want to run no matter what should be there (eg closing streams)

Object Serialization

- Wouldn’t it be nice to be able to just write or read an entire object at a time, and not worry about the underlying messy parts?
- All you have to do is implement java.io.Serializable.
- That’s It! Done!

Object Serialization (cont’d)

- writeObject() will throw an Error if the object passed to it is not Serializable.
- You can control serialization by implementing the Externalizable interface.
- readObject() returns something of type Object, so it needs to be cast.

Object Serialization (cont’d)

- If a field is transient, it’s value is not persistent when the object is serialized.
  - This is useful if your object has members that are not necessary for the object to be reconstructed.
- Overall, this is very useful for high-level object storage.

java.io.File

- The File class is very useful for information about files and the directory structure.
- Constructer takes a String for the name
- Useful methods:
  - .exists()
  - .isDirectory()
  - .listFiles() - Lists the files if it’s a directory

java.io.File (cont’d)

- .canRead() / .canWrite() - check permissions
- File.listRoots() - returns an array of the roots of a file system.
- .length()
- .delete()
- Look in the documentation for the rest.

java.io.RandomAccessFile

- *Not* a stream file.
- There is a file “pointer” which you can use to read and write to random place in the file.
- You can read and write only primitive data types - writeByte(), writeInt(), writeBoolean()
- It’s faster to jump between points in the file, than search an entire stream file.