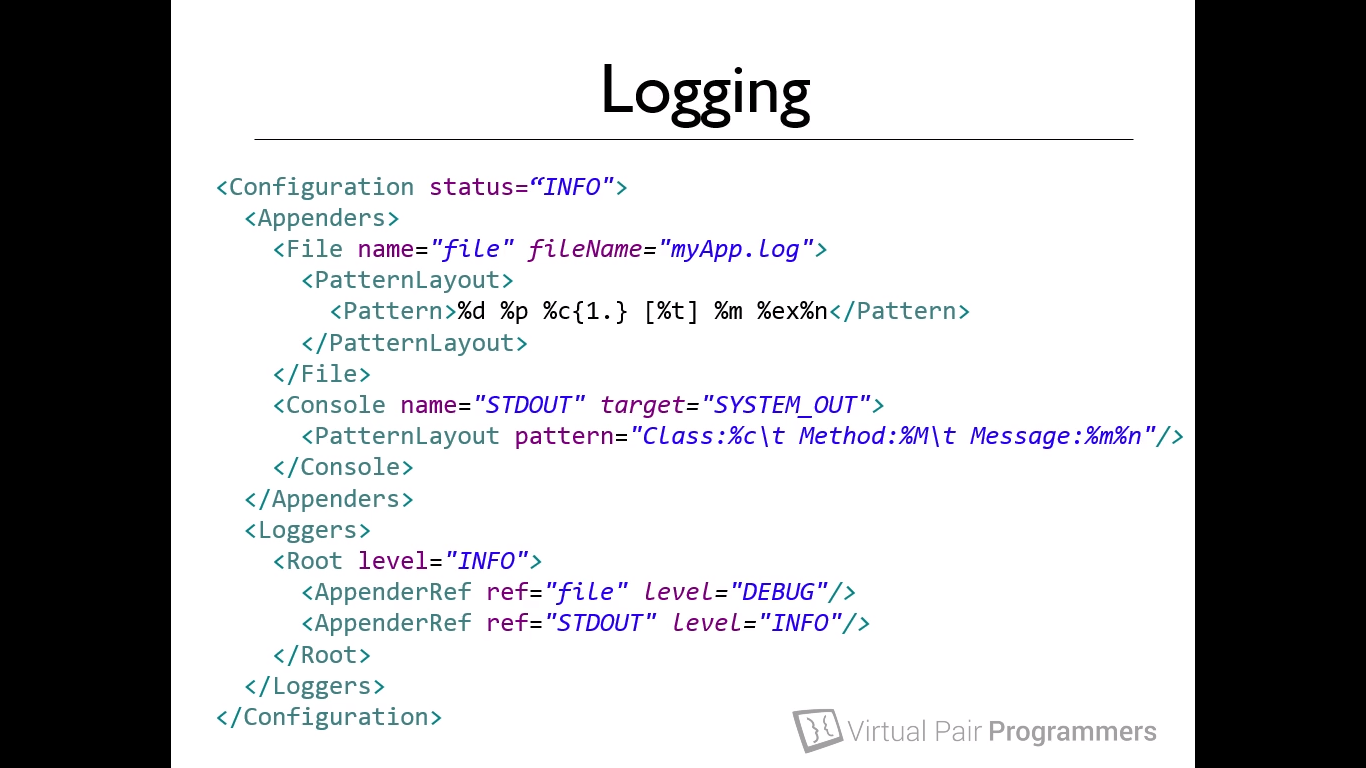
**Logging**

Two main concepts in logging –

* Loggers – which generates logging events
* Appenders – This takes the logging event and writes to console/files whichever is configured.

By default – only those error are produced whose level are either ERROR of higher such as FATAL. So we need a configuration file to manage the events and override the default setting provided by Logging framework.



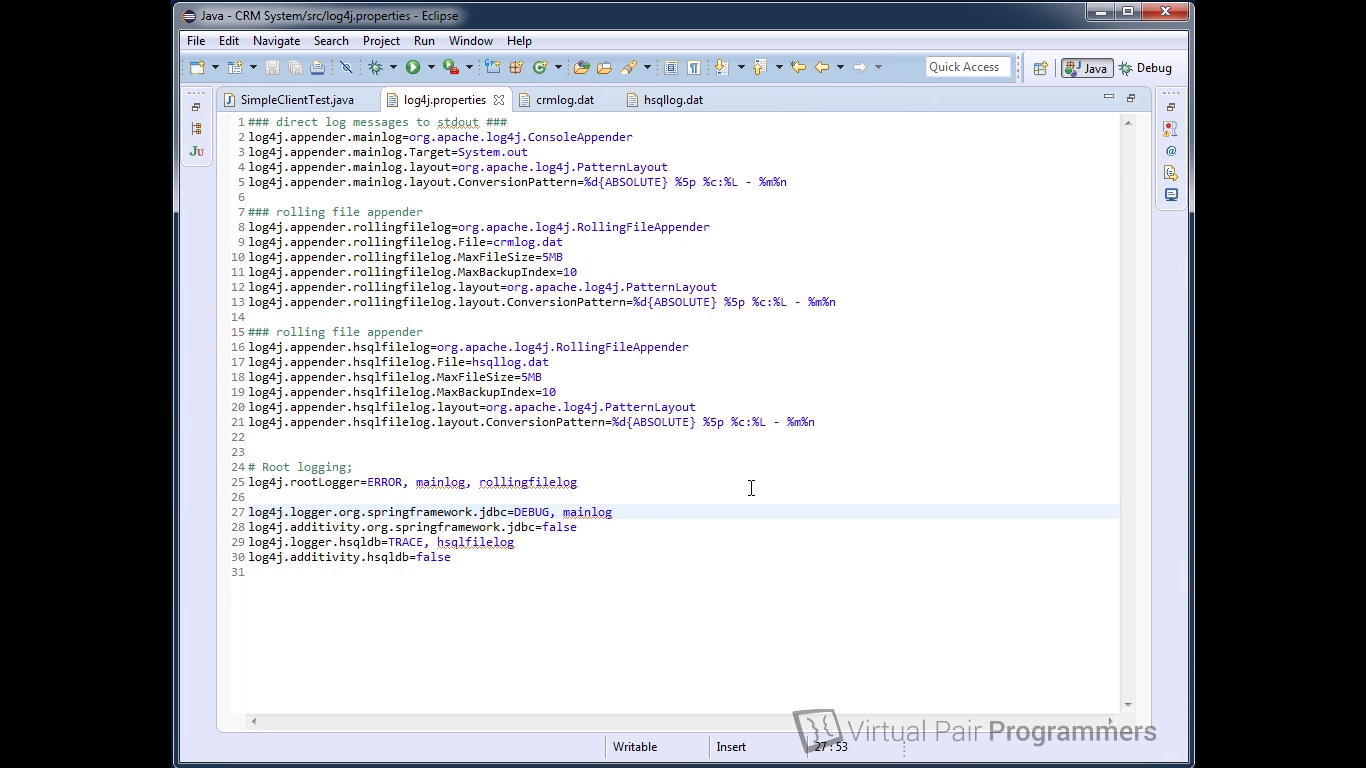
Simple xml configuration file – It has two appenders, one for the file and one for the standard console output. Root level severity level is set to INFO. Also we can override the logging level in Loggers setting which is individually set for file as well as STDOUT.

Note – If we don’t define any logging level like here in top configuration INFO, the default will be ERROR only and all the logging levels below the error will not be outputted (if no levels are specified in AppenderRef also). There must be somewhere log levels should be specified.

A more standard way of outputting logs is to have a properties file and defining the log properties.

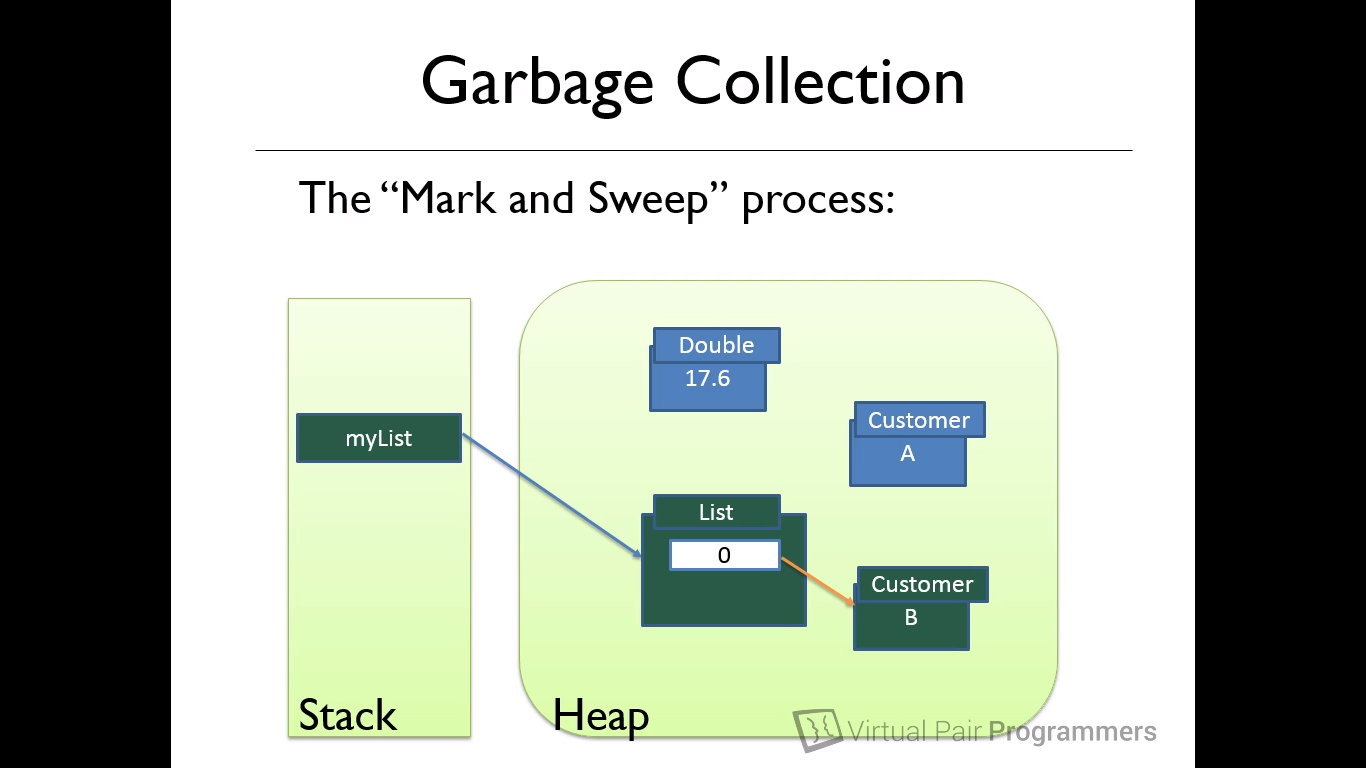
Below files have 3 appenders and different loggers

* Root logger will output all error messages to the console and rollingFileAppender.
* All springframework.jdbc will go to console
* Hsqldb.\* class output will go to the appender hsqlfile log with level TRACE.

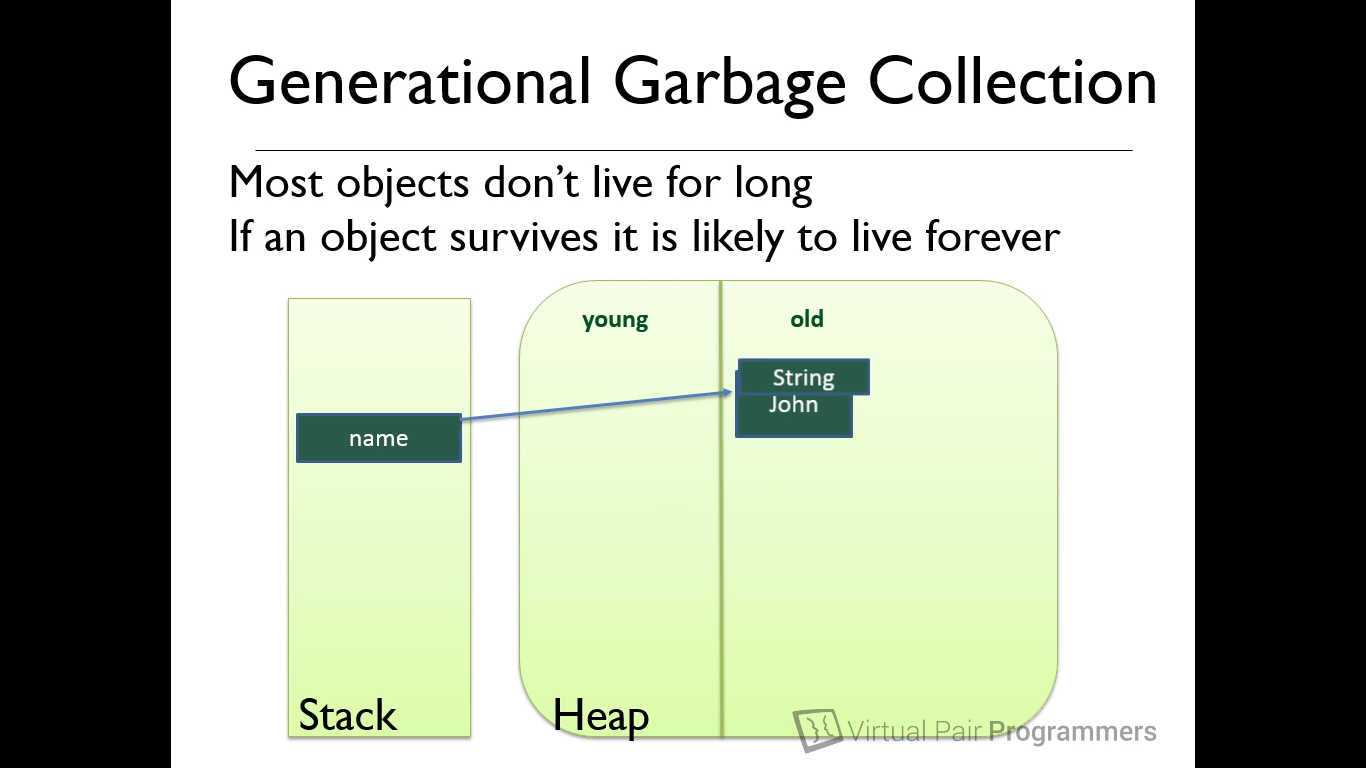


**Garbage Collection –**

Garbage collection does not work by collecting all the garbage objects that is objects that are not referenced by variables in stack but it’s the other way. It retains and saves the object by marking if the objects are referenced by stack variables. In below picture List and Customer B is saved as they are marked and Double and Customer A is garbage collected. After GC the saved objects are then moved to some contiguous memory space (compacting process) in heap to avoid disk fragmentations.



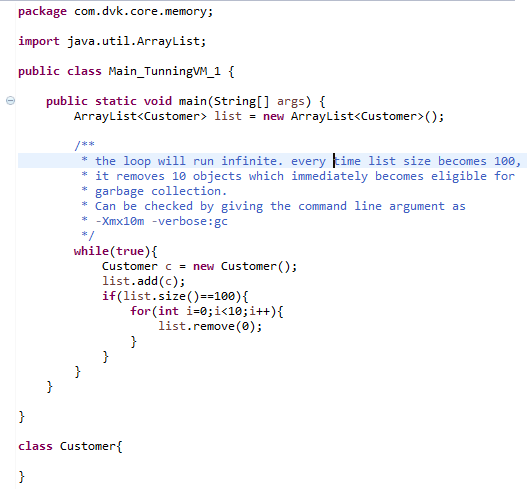
Now a typical GC operation must pause the program for fraction of time to mark the objects but this is not acceptable. To avoid this, it uses a technique called **Generational Garbage Collection**.



Here in this process, Heap memory is divided into young and old generation area. Young generation area is very small and all the new objects are created in this area only. So when GC happens, there are relatively very small objects to mark and it is very fast so we do not notice the pause in the program. The objects are survived are then moved to old generation and young generation is freed up of memory. The GC on young generation is called **Minor collection**. GC can also run on old generation but only if its needed which is called Major collection. This is relatively very slow as there are lot of objects to mark.

Virtual machine tuning –

1. Tuning virtual machine means providing command line arguments to VM to override default values of jre.
2. –**Xmx** = to set maximum heap size.
3. –**Xms** = to set the starting heap size, size will gradually increase
4. –**Xmx512m** –Xms150m = will have 150mb of starting and 512 mb of max heap size.
5. –XX:**MaxPermSize**=256 = set the size of perm gen space – this is removed from java 8 as java 8 does not have permgen space concept anymore.
6. –**verbose**:gc = prints the console when gc takes place



1. –**Xmn** = set the size of young generation
2. –XX:**HeapDumpOnOutOfMemory** = creates a heap dump when application runs out of memory.
3. Types of Garbage collector –
   1. Serial **(-XX:+UseSerialGC**) – only single thread will execute the garbage collection and application will be paused while doing the garbage collection process.
   2. Parallel **(-XX:+UseParallelGC**)
   3. Concurrent – **(-XX:+UseParallelGC**) Application is not paused (practically not possible, app will pause when process of object marking and app resumes when sweep phases starts).
4. We can check the type of garbage collection in our machine from the command

- XX: PrintCommandLineFlags.

**Hunting for memory leaks**