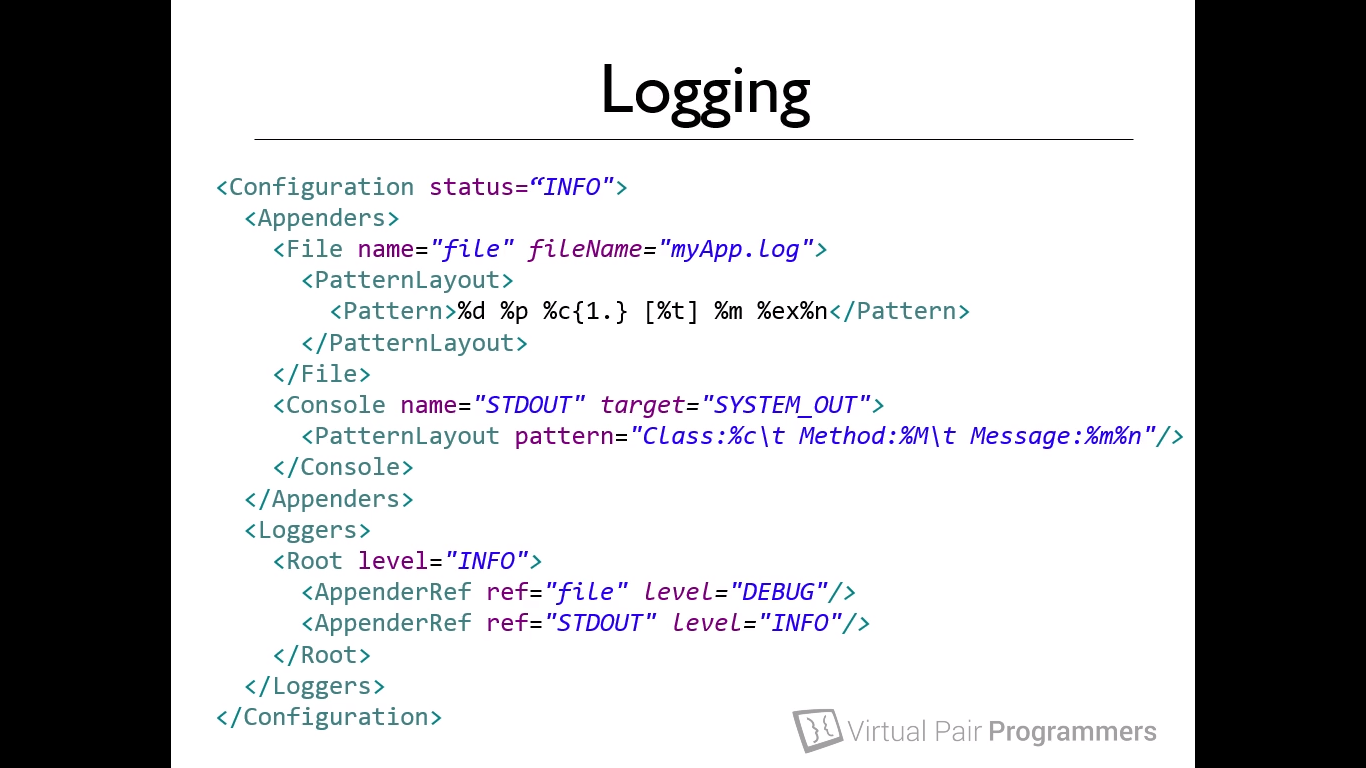
**Logging**

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.



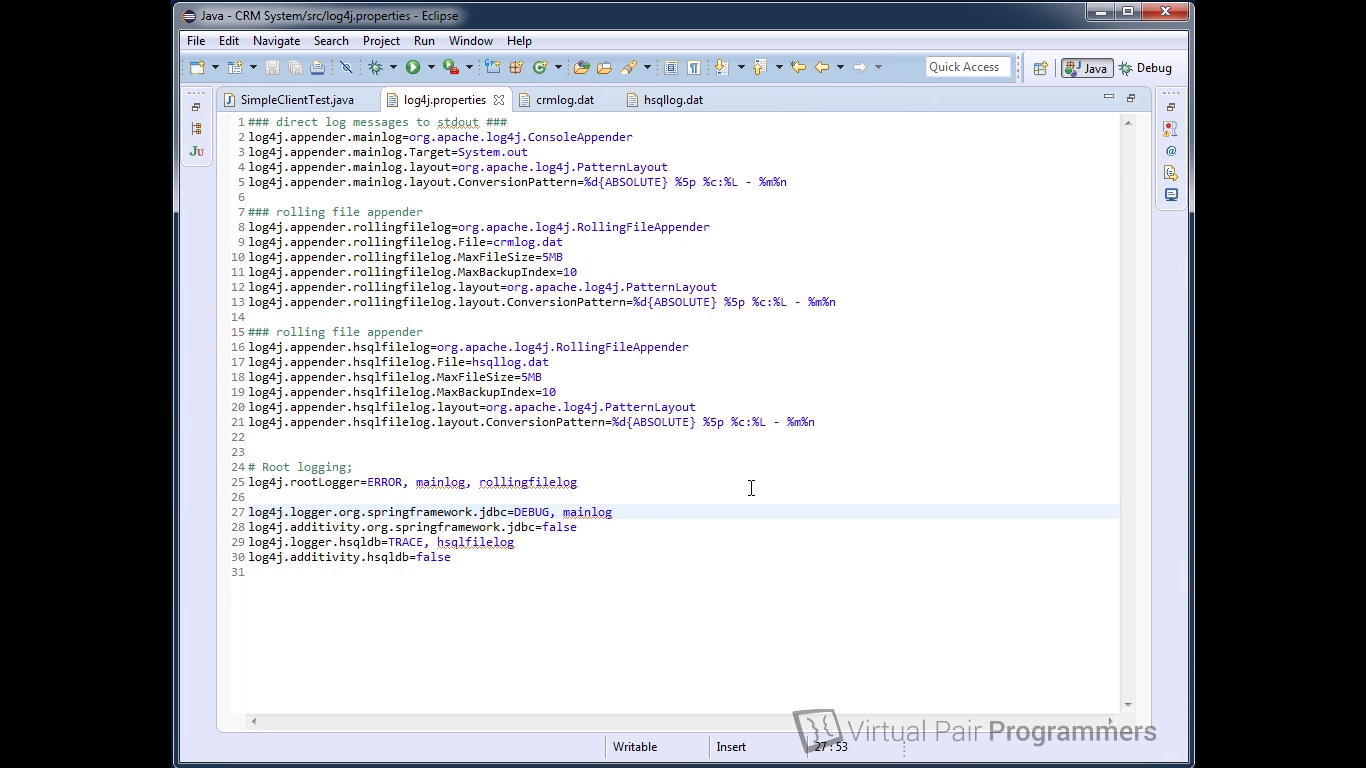
Simple xml configuration file – It has two appenders, one for the file and one for the standard console output. Root level severity level is set to INFO. Also we can override the logging level in Loggers setting which is individually set for file as well as STDOUT.

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.

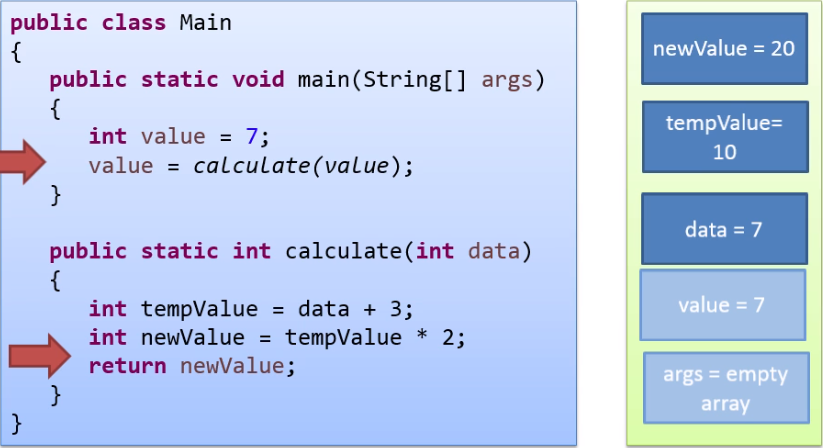
Below files have 3 appenders and different loggers

* 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.

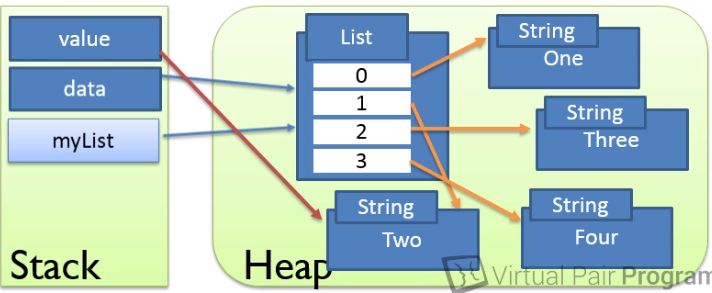
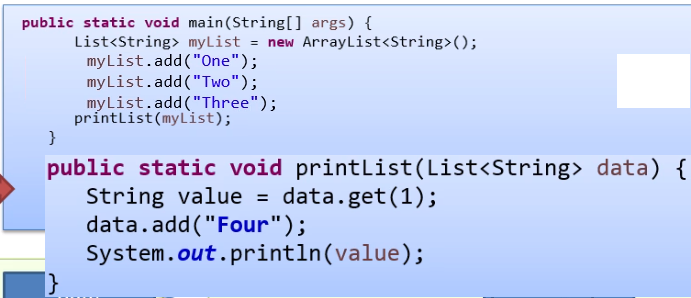


**Memory Fundamentals –**

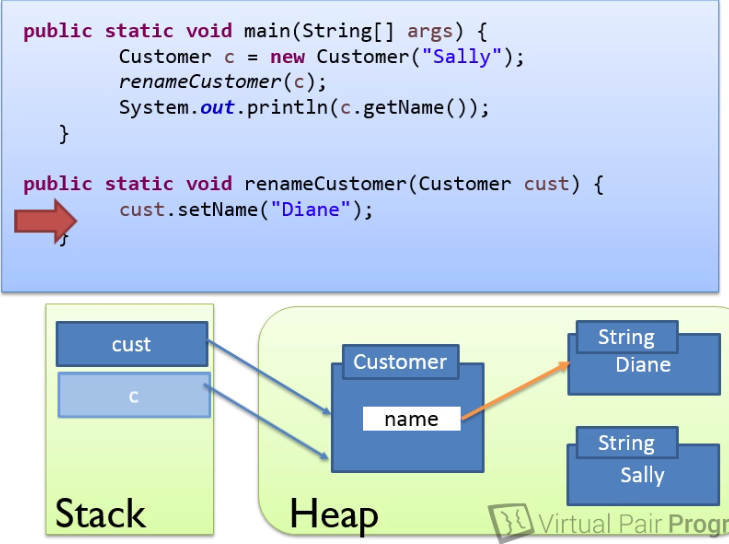
Stack – every local variable resides in a stack. A thread has its own stack and all the variables are only visible for that thread. Once program terminates, the final closing braces, all the local variables are popped off from the tread. When we call a function with primitive, a copy of the variable is passed in the method calculate.



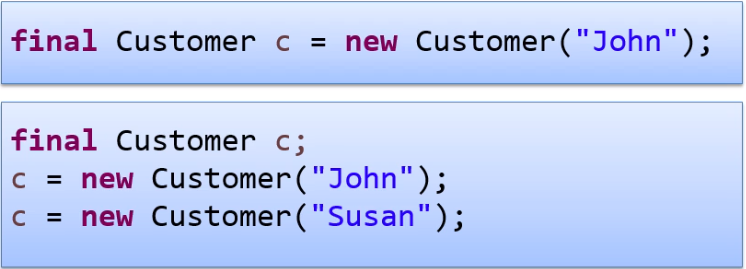
Heap – All the objects will be stored on the heap and the local variables and reference will be stored in the stack. In the printList function, the variable data will have the copy of the value of variable myList which is the address of list. ***So all variable are passed by value.***



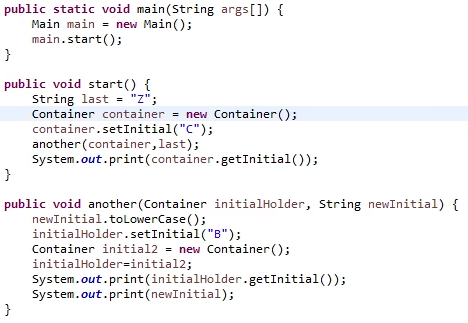
Passing value into function – **When objects are passed into methods, the REFERNECE to the object is passed BY VALUE**. So yes, actually all variables are passed by value. And it is not true that object is passed by reference.



Final – The below code will give compilation error as we can’t change the value of the variable which is declared final. Note – value means we can’t change the reference of the variable to the new address to Susan. But, but we can change the underlying value of the object by setting the name to Susan and the values will be changed simple. So there is not exact constant correctness in java and we need to make sure that we do not make unnecessary changes when we pass an object to different methods as final won’t stop object from changing internally.



Output test –



AZB

**Escaping references –**

The problem where we return a pointer to the object in the getters method is called escaping reference. So in getters methods, we directly return the variable which holds the object of say a List. Problem – the list can be modified as we have direct pointer to the memory and can lead to corruption.

Solutions –

1. Collections - we can use a Collection.unmodifiableList () method in case of list collection to return a copy of list which is not modifiable.
2. Userdefined class objects – for getters returning the reference to the objects, we can use a **copy constructor** which will take the object and make copy of the object and return the copy and not the actual object. This is a partial fix as this can lead to confusion if we modify data by getting the customer back and will not reflect the changes as the reference is different. Another solution is to return an Interface which contains only getters method and not the setter method on the object. So in a way all the setters’ methods will not be visible.



1. Getters for all the immutable objects like String, Integers and other wrappers are immutable and hence will not lead to escape reference.

**Garbage Collection –**

Internalized strings – JVM/Java has a concept of String immutability, means String objects once created, cannot be changed. So by using this phenomenon, if in programs declaring the same String object value (String literals only) twice, the JVM will not create a different object but just point it to the already created String object. This is known as **Internalized strings**. This saved heap memory by reusing the object as String is immutable.

Once the object is eligible for GC, before removing the object, garbage collector will run **finalize** () method of that object. It is not guarantee that GC will always happen and it will happen on need basis, so it is not good to put code in finalize that we want it to run evertime. – This method has no use.

Mark and Sweep - 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. After GC the saved objects are then moved to some contiguous memory space (compacting process) in heap to avoid disk fragmentations.

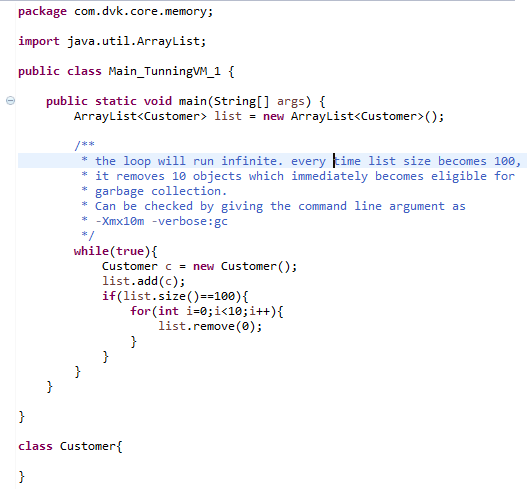
Now a typical GC operation must pause the program for fraction of time to mark the objects but this is not acceptable. To avoid this, it uses a technique called **Generational Garbage Collection**.

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 (only on the young generation), there are relatively very small objects to mark and it is very fast so we do not notice the pause in the program. All objects that 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 it’s needed which is called Major collection. This is relatively very slow as there are lot of objects to mark.

* The memory areas are visible with the help of tool *Visual VM* from inside JDK/bin folder and by installing a *Visual GC* plugin inside the visual vm.
* *Heap Dump* from monitor tab to obtain the graph.
* Download *Memory Analyser* (MAT) to analysis the heap dump file.
* *Jmeter* to load test the application

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. So running the below program with small memory and with verbose gc option, we can see the GC process print on console.



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

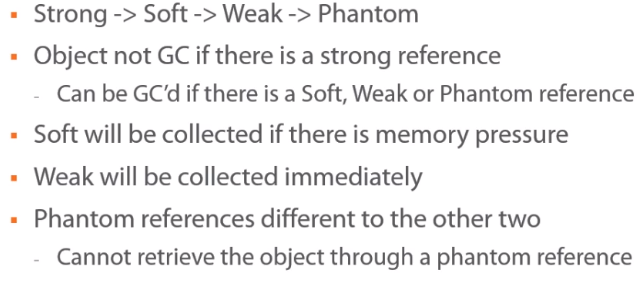
- XX: PrintCommandLineFlags.

**Hunting for memory leaks – Google**

**Weak and Soft references –**

Weak references – the objects that may not survive GC even if the reference to the objects is available. **WeakhashMap** is an example as it maps the key and value pair with weak relations.

Soft reference – the objects will survive the GC but will be removed from the memory if there is low memory issues. Suitable for caching.



Demo example for weak reference – we are doing null for all the strong references as only then the GC demo will occur on weak reference.

