**## Advanced Java Study Guide**

You should be able to explain and apply the following topics:

|  |  |
| --- | --- |
| **### Threads and Garbage Collection** | |
| Thread states- | 1. **New** - When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it’s code is yet to be run and hasn’t started to execute. 2. **Runnable** - A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.      1. **Blocked** - **Temporarily inactive**; the thread is still alive, but is currently not eligible to run. Could be waiting to acquire a lock. It’s the responsibility of the thread scheduler to reactivate and schedule a blocked thread. 2. **Waiting - Temporarily inactive;** when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state. It’s the responsibility of the thread scheduler to reactivate and schedule a waiting thread. 3. **Timed Waiting** - A thread lies in timed waiting state when it calls a method with a timeout parameter. A thread lies in this state until the timeout is completed or until a notification is received 4. **Terminated** -  thread terminates because of either of the following reasons:   - Because it exists normally. This happens when the code of thread has entirely executed by the program.  - Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception. |
| thread | A subset of a process that is also an independent sequence of execution. All threads share the same heap.  **Problems with Threads:**  Race Conditions  Thread Starvation  Producer-Consumer Problem |
| Producer-Consumer Problem | **Producer** – an entity which creates or populates an area with data, known as the queue.  **Consumer** – an entity which takes and removes data from an area, known as the queue.  **Problem:** The producer should produce data when the queue is not full. If the queue is full, then the produce should not be allowed to put any data into the queue. The consumer should consume data only when the queue is not empty. If the queue is empty, then the consumer should not be allowed to take any data from the queue.  **Solution:** use of the wait() and notify() methods can be used to communicate between produce and consumer threads.  The wait() method is used to pause the producer or consumer thread based on the size of the queue.  The notify() method sends a notification to the waiting thread |
| Race Conditions | The result of a transaction is dependent on other threads access to it. Race conditions can be solved using synchronization, to limit access to a resource to only one thread at a time. |
|  |  |
| Lifecycle of a thread | **New** - he thread is in new state if you create an instance of Thread class but before the invocation of start() method.  **Runnable** - The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.  **Running** - The thread is in running state if the thread scheduler has selected it  **Non-Runnable (Blocked)** - This is the state when the thread is still alive, but is currently not eligible to run.  **Terminated** - A thread is in terminated or dead state when its run() method exits. |
| - Creating threads | \* Extending the Thread class  \* Implementing the Runnable Interface  \* Creating a Lambda |
| - Thread class & thread methods | Some important thread methods include:  - getters and setters for id, name and priority  - interrupt () - used to explicitly interrupt a thread’s operations  - isAlive (). islnterrupted () , isDaemon() — used to test the state of a thread  - join () - used to wait for the thread to finish execution  - start () - used to begin thread execution after instantiation.  Additionally, some important static methods include:  - Thread. currentThread () - returns the thread that is currently executing  - Thread. sleep (long millis) — causes the currently executing thread to temporarily stop for a specified number of milliseconds. |
| - Runnable interface |  |
| - Deadlock and ways to prevent it | **Deadlock** - All processes are in the waiting state. a state in which each member of a group of actions, is waiting for some other member to release a lock. A stalemate.    **Starvation** - when “greedy” threads make shared resources unavailable for long periods. |
| - Livelock | When two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently. |
| - How to invoke Garbage Collection | **Garbage Collection** - Main objective of Garbage Collector is to free heap memory by destroying unreachable objects. An object is said to be unreachable if it doesn’t contain any reference to it.  System.gc() - We call the GC.  Runtime.getRuntime().gc() - We call the GC.  System.runFinalize()  Runtime.getRuntime().runFinalize() |
|  |  |
| **### Reflections API** | |
| - Benefits and purpose of Reflection API | Used to examine or modify the behavior of methods, classes, interfaces at runtime.  - Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.  - Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.  **Advantages of Using Reflection:**  Extensibility Features: An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.  Debugging and testing tools: Debuggers use the property of reflection to examine private members on classes.    **Drawbacks**:  Performance Overhead: Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.  Exposure of Internals: Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform. |
| - Classes and interfaces |  |
| - Class |  |
| - Method |  |
| - Modifier |  |
| - Parameter |  |
| - Constructor |  |
|  |  |
| **### Java 8** | |
| - Functional interfaces | an interface that contains only one abstract method. |
| - Lamdba functions | - Enable to treat functionality as a method argument, or code as data.  - A function that can be created without belonging to any class.  - A lambda expression can be passed around as if it was an object and executed on demand. |
|  |  |
| **### JUnit** | |
| - Test-driven development (TDD) | The TDD process consists of writing unit tests first, before the application code has been written. Then, code can be written to make the test pass, and the process can be completed for each piece of functionality required. |
| - JUnit annotations |  |
| - @Test | declares a method as a test method |
| - @Before | declares a setup method that runs before each test method is run |
| - @After | declares a ‘tear-down’ method that runs after each test method |
| - @BeforeClass | declares a setup method that runs once, before all other methods in the class |
| - @AfterClass | declares a ‘tear-down’ method that runs once, after all other methods in the class |
| - @Ignore | declares that the proceeding test will not be run.  Used to check other test cases within a test class before refactoring individual test cases. |
| - Order of execution of these annotated methods | BeforeClass  Before  Test  After  Before  Test  After  AfterClass  Ignore |
| - Assert class methods | **assertEquals()** – Test to see if a return value is the same as an expected result  **assertNotEquals()** – Test to see if return value differs from an expected result  **assertNull()** – Test to see if an object/variable/etc… reference is null  **assertNotNull()** – Test to see if an object/variable/etc… has a non-null reference  **assertArrayEquals()** – Test to see if the values of an array match a provided array.  **assertTrue()** – Test to see if something is true  **assertFalse()** – Test to see if something is false |
| - Testing best practices |  |
| - |  |
| - Writing testable code |  |
| - Mocking | Mock is an Object that clone the behavior of a real object. It is basically used in Unit Testing by testing the isolated unit even when Backend is not available. |
| - Measuring code coverage |  |
| - Externalize data |  |
|  |  |
| **### Log4j** | |
| - Benefits / purpose of logging | * Recording unusual circumstances or errors that may be happening in the program * Getting the info about whats going in the application |
| What is Log4J? | reliable, fast and flexible logging framework for Java |
| - Log4j configuration | **Configuration Programmatic implementation** can be done using:  - configuration file written in XML, JSON, YAML, or properties format  - A factory design pattern and creating a ConfigurationFactory with Configuration implementation  - Calling APIs exposed in the configuration interface  - Calling methods on the internal Logger class |
| - Log4j logging levels and threshold |  |
| - ALL | All 8 levels |
| - Trace | finer-grained informational events than DEBUG |
| - DEBUG | designates informational events that are most useful to debug an application |
| - INFO | informational messages that highlight the progress of the application at the coarse-grained level |
| - WARN | designates potentially harmful situations |
| - ERROR | designates error events that might still allow the application to continue running |
| - FATAL | severe error events that presumably lead the application to abort |
| - OFF | highest possible level, intended to turn off logging |
| **Other Terms worth noting** | |
| Time splicing | performing short bursts of different processes and switching between them rapidly. |
| Concurrency | refers to breaking up a task or piece of computation into different parts that can be executed independently, out or order, or in partial order without affecting the final outcome. |
| Thread priority | Can have input on the priority of threads. 1 (min priority) to 10 (max priority) |
| Logging | The act of recording granular events within an application, such as transactions with a database, exceptions or debugging events. Generally logs are maintained by writing them to a file. Without logs, it is difficult to impossible to know what went wrong when an application crashes, or monitor the performance of an application. |
| Log4j class architecture | **Logger** – logs the messages  **Appender** – publishes logs to destination(s)  **Layout** – formats logging information  **Configuration** – stores settings  **Filter** – used to filter logs that do not meet some threshold or are not required. |
|  |  |
|  |  |
|  |  |