**Multithreading:**

Multithreading typically employs internal mechanisms, such as locks, atomic operations, or lock-free algorithms, to handle concurrent modifications and ensure thread-safety.

**Java Synchronization Language Features:**

Synchronized methods, blocks

Volatile & atomic variables

Locks - deadlock/livelock

Thread-safe collections - ConcurrentHashMap (java.util.concurrent)

CopyOnWriteArrayList

BlockingQueue

Thread synchronization utilities - CountDownLatch (java.util.concurrent)

CyclicBarrrier

Semaphore

Exchanger

**Synchronizing Resource Access:**

* Use the right synchronization level.
* Minimize the synchronized block/method scope.
* Use read-write locks for read-mostly scenarios.
* Consider lock striping.
* Utilize concurrent data structures.
* Use thread-local variables.
* Prefer immutable objects.
* Leverage lock-free algorithms.
* Test and profile.

**Concurrent Collections:**

* ConcurrentHashMap
* ConcurrentLinkedQueue
* ConcurrentSkipListMap - provides sorted key-value mappings
* CopyOnWriteArrayList - makes fresh copy of array
* BlockingQueue
  + threads can be blocked from performing operations (useful for producer-consumer scenarios)
  + LinkedBlockingQueue
  + ArrayBlockingQueue

**Java Synchronizers & Lock:**

* These are the mechanisms provided by java.util.concurrent
  + Lock Interface
    - Alternative to synchronized blocks
    - Provides lock(), unlock()
  + ReentrantLock
    - Fully-featured implementation of Lock interface
    - Supports reentrant behaviour
      * Reentrant - pause the task, perform other process and resume previous one.
    - Allows multiple levels of locking
  + Condition Interface
    - Thread sould met conditions before performing task.
  + ReentrantReadWriteLock
    - Multiple threads can read a shared resource as long as no thread is holding write lock.
  + Semaphore
    - Control access to resources with limited capacity/to limit concurrent access to certain no.of threads.
  + CountDownLatch
    - Allows one/more threads to wait until some operations complete. It is initialized with a count.
    - countDown() - decrement the count.
    - Await() - to wait until count reaches to zero
  + CyclicBarrier
    - Allows set of threads to wait until all threads reach a common point.
    - Await() is used to wait

**Thread Pooling Techniques:**

* Executor Framework - thread management & task execution
* ThreadPool Executor - create & manage thread pools
* ScheduledThreadPoolExecutor - schedules tasks to run periodically or with delay
* Executors utility class - has factory methods to create different types of thread pools
* ForkJoinPool - a thread pool, optimized for divide & conquer algorithms & recursive algorithms
* Work stealing - pool implementation technique. Ensures all threads are busy with work.
* Custom thread pool - to create custom thread pool

**Executor Framework:**

* Create executor
* Define task
* Submit tasks for execution
* Shutdown executor
* Specifying thread pool size
* Thread pool configuration
* Handling task results

**Use Pool Effectively:**

* Determine optimal pool size
* Reuse threads
* Use thread pools for long running tasks
* Use proper task abstractions
* Avoid blocking operations
* Handle exceptions properly
* Monitor & tune the pool
* Gracefully shutdown the pool
* Consider specilized pools
* Test & benchmark

**Advanced Java**

**Inner Class/Nested Class Types:**

Like nested class, we also have nested enums, nested interfaces.

* Member inner class - class inside a class
* Static nested class - static class inside a class
* Local inner class - class inside a method
* Anonymous inner class - inside a method

**Java API Techiniques:**

* Console class - used to interact with console/command line
* StringBuilderClass - used to manipulate strings
* Formatting - used to format data
  + Regular Expressions
  + Number formatting
    - DecimalFormat class
  + Date & Time formatting
    - SimpleDateFormat
  + Currency Formatting
    - NumberFormat class
  + String.format()
* Localization
  + Used to provide regional & cultural preferences for formatting & displaying data
  + Locale class is used
* Resource bundles
  + Used for internalization & localization purposes
  + Allows to store locale specific resources like strings, messages, lables, configurations etc..
  + ResourceBundle Class is used
* Logging
  + Used to record application events & troubleshoot issues
  + Loggers are entry points to java logging framework, these are static variables within classes & used to log messages. Loggers are named hierarchially & follows dot separated naming convention.
  + logging.properties holds the logging configuration. Configuration file allows devs to specify properties such as log levels, handllers, formatters & logging destinations
  + Log Levels
    - FINEST - for tracing & debugging & provides high detailed debugging info
    - FINER - provides fine-grained debugging info
    - FINE - provides less detailed debugging info
    - CONFIG - for providing configuration related info
    - INFO - progress/state of application. Used for important app events
    - WARNING - indicated potential issues
    - SEVERE - indicates critical errors
    - DEBUG - captures detailed info about the execution flow of application
  + Best practices
    - Use appropriate log level
    - Include relevant info
    - Use parameterized logging
    - Log exceptions properly
    - Enable logging only when necessary
    - Customize log output format, destinations & log handlers (use configuration files)
* Log4j
  + Logging levels & loggers
    - Extra logging levels like
      * TRACE
      * DEBUG
      * INFO
      * WARN
      * ERROR
      * FATAL
  + Appenders
    - Responsible for delivering log messages to files, db’s, consoles, remote servers
  + Layouts
    - Defines format of log messages
  + Easy configuration
    - Configured through a configuration file names **log4j2.xml** or **log4j2.properties**
  + Asynchronous logging
    - It separates the log message generation from actual writing to log destination, which allows applications to continue its execution without waiting for log messages to written.
* Garbage Collection
  + Object lifecycle
    - Creation
    - Reachability
    - Marking
    - Sweep & Deallocate
  + Generational Collectors
    - It is a type of garbage collector that divide the heap into generations based on age of objects
    - Heap is divided into 3 main generations.
      * Young generation - new objects are allocated & aged here
      * Old generation -
      * Permanent generation - metadata about classes, method bytecode
      * Metaspace (after java8) - replacement of permanent generation
        + Dynalically grows & shrinks based on application needs

**Mockito**

**Stub:**

* Stubs are placeholder of actual objects
* Stub Lifecycle
  + Prepare object to be tested
  + Test the functionality
  + Using asserts to test
  + Clean up resources