3. How to create a new thread (including Thread Pool approach)?

There are several ways:

- **Extend Thread class**: Create a subclass of Thread and override run() method
- **Implement Runnable**: Create a class implementing Runnable interface and pass it to Thread constructor
- **Implement Callable**: Similar to Runnable but can return a result and throw exceptions
- **Thread Pool**: Use ExecutorService (like newFixedThreadPool, newCachedThreadPool) to manage a pool of reusable threads, which is more efficient for handling many short-lived tasks

4. Difference between Runnable and Callable?

Runnable has a run() method that returns void and cannot throw checked exceptions. **Callable** has a call() method that returns a result (generic type) and can throw checked exceptions. Callable is used with ExecutorService and returns a Future object to retrieve the result.

5. What is the difference between t.start() and t.run()?

t.start() creates a new thread and executes the run() method in that new thread - true concurrent execution. **t.run()** simply calls the run() method in the current thread, just like any regular method call - no new thread is created, no concurrency.

6. Which way of creating threads is better: Thread class or Runnable interface?

Runnable interface is better because Java supports single inheritance only, so implementing Runnable allows your class to extend another class if needed. It also promotes better design by separating the task (what to run) from the execution mechanism (thread). This makes code more flexible and reusable.

7. What are the thread statuses?

Java threads have six states:

- **NEW**: Thread created but not yet started
- **RUNNABLE**: Thread executing or ready to execute

- **BLOCKED**: Thread waiting for a monitor lock
- **WAITING**: Thread waiting indefinitely for another thread's action
- **TIMED_WAITING**: Thread waiting for a specified time
- **TERMINATED**: Thread completed execution

8. Demonstrate deadlock and how to resolve it in Java code

Deadlock occurs when two or more threads wait for each other to release locks, creating a circular dependency. Example: Thread A holds Lock1 and waits for Lock2, while Thread B holds Lock2 and waits for Lock1.

Resolution strategies: Always acquire locks in the same order across all threads, use timeouts with tryLock(), use higher-level concurrency utilities, or avoid nested locks when possible.

9. How do threads communicate with each other?

Threads communicate through:

- **wait(), notify(), notifyAll()**: Used with synchronized blocks for inter-thread communication
- **Volatile variables**: Ensures visibility of changes across threads
- **BlockingQueue**: Producer-consumer pattern implementation
- **CountDownLatch, CyclicBarrier, Semaphore**: Coordination utilities
- **Locks and Conditions**: More flexible than synchronized blocks

10. What's the difference between class lock and object lock?

Object lock (instance lock) is acquired on a specific object instance using synchronized instance methods or synchronized(this). Different instances have different locks. **Class lock** (static lock) is acquired on the Class object using synchronized static methods or synchronized(ClassName.class). There's only one class lock per class, shared across all instances.

11. What is join() method?

The join() method makes the calling thread wait until the thread on which join() is called completes execution. For example, if main thread calls thread1.join(), the main thread will pause and wait for thread1 to finish before continuing. It's useful for ensuring tasks complete in a specific order.

12. What is yield() method?

The yield() method is a hint to the thread scheduler that the current thread is willing to yield its current use of the processor. The scheduler may ignore this hint. It moves the thread from running to runnable state, allowing other threads of same or higher priority to execute. However, it's rarely used in practice as its behavior is platform-dependent.

13. What is ThreadPool? How many types of ThreadPool? What is the TaskQueue in ThreadPool?

ThreadPool is a collection of reusable threads that execute submitted tasks. Types include:

- **FixedThreadPool**: Fixed number of threads
- **CachedThreadPool**: Creates threads as needed, reuses idle ones
- **SingleThreadExecutor**: Single worker thread
- **ScheduledThreadPool**: For delayed/periodic execution

TaskQueue (BlockingQueue) stores submitted tasks waiting for execution when all threads are busy. Different pools use different queue implementations (LinkedBlockingQueue, SynchronousQueue, etc.).

14. Which Library is used to create ThreadPool? Which Interface provides main functions of thread-pool?

The **java.util.concurrent** package provides thread pool functionality. The **ExecutorService** interface provides the main functions for thread pool management, including methods for submitting tasks, shutting down the pool, and managing execution. The **Executors** class provides factory methods to create different types of thread pools.

15. How to submit a task to ThreadPool?

Use ExecutorService methods:

- **execute(Runnable)**: Submits a Runnable task, returns void
- **submit(Callable)**: Submits a Callable task, returns Future for result retrieval
- **submit(Runnable)**: Submits a Runnable, returns Future (result will be null)
- **invokeAll()**: Submits collection of Callable tasks
- **invokeAny()**: Submits tasks, returns result of one that completes successfully

16. What is the advantage of ThreadPool?

Advantages include:

- **Resource management**: Controls number of concurrent threads, prevents resource exhaustion
- **Performance**: Reuses threads instead of creating/destroying them repeatedly
- **Task management**: Provides queue for pending tasks
- **Responsiveness**: Separates task submission from execution
- **Statistics**: Can track completed tasks, active threads
- **Flexibility**: Easy to configure thread count, queue size, rejection policies

17. Difference between shutdown() and shutdownNow() methods of executor

shutdown() initiates an orderly shutdown where previously submitted tasks are executed, but no new tasks are accepted. It doesn't wait for tasks to complete and returns immediately.

shutdownNow() attempts to stop all actively executing tasks, halts processing of waiting tasks, and returns a list of tasks that were awaiting execution. It interrupts running threads but doesn't guarantee they'll stop.

18. What is Atomic classes? How many types and main methods?

Atomic classes provide lock-free, thread-safe operations on single variables using compare-and-swap (CAS) operations. Types include:

- **AtomicInteger, AtomicLong, AtomicBoolean**: For primitive types
- **AtomicReference**: For object references
- **AtomicIntegerArray, AtomicLongArray**: For arrays
- **AtomicStampedReference, AtomicMarkableReference**: For solving ABA problem

Main methods: get(), set(), getAndSet(), compareAndSet(), getAndIncrement(), incrementAndGet(), getAndAdd(), addAndGet()

When to use: For simple counters, flags, or single-variable updates where synchronized blocks would be overkill. They're more efficient than locks for simple atomic operations.

19. What are concurrent collections? List some thread-safe data structures

Concurrent collections are thread-safe data structures designed for concurrent access without external synchronization. Examples:

- **ConcurrentHashMap**: Thread-safe HashMap with better concurrency than Hashtable
- **CopyOnWriteArrayList**: Thread-safe ArrayList, optimized for read-heavy scenarios
- **ConcurrentLinkedQueue**: Non-blocking thread-safe queue
- **BlockingQueue implementations**: LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue
- **ConcurrentSkipListMap/Set**: Thread-safe sorted map/set
- **ConcurrentLinkedDeque**: Thread-safe double-ended queue

20. What kind of locks do you know? Advantages of each?

ReentrantLock: More flexible than synchronized, supports fairness, tryLock with timeout, interruptible lock acquisition, and multiple condition variables.

ReadWriteLock: Allows multiple readers or one writer, improving performance for read-heavy scenarios.

StampedLock: Optimized version of ReadWriteLock with optimistic read mode for better performance.

Synchronized: Built-in, automatic lock release, simpler syntax, but less flexible.

Each has trade-offs: synchronized is simpler, ReentrantLock more flexible, ReadWriteLock better for read-heavy loads, StampedLock best performance but more complex.

21. What is Future and CompletableFuture? Main methods of CompletableFuture

Future represents the result of an asynchronous computation. You can check if it's done, wait for completion, and retrieve the result. However, it has limitations: blocking get(), no chaining, no error handling.

CompletableFuture is an enhancement that supports:

- Non-blocking operations
- Chaining: thenApply(), thenAccept(), thenRun()
- Combining: thenCombine(), thenCompose()
- Error handling: exceptionally(), handle()
- Async execution: supplyAsync(), runAsync()
- Waiting for multiple: allOf(), anyOf()

It enables true asynchronous, non-blocking programming with functional-style composition of async operations.