**MultiThreading**Multithreading is the ability of a CPU to provide multiple threads of execution concurrently within a single process. This allows for more efficient use of resources and better performance in applications.

In a single-core environment, Java’s multithreading is managed by the JVM and the OS, which switch between threads to give the illusion of concurrency.

When a Java program starts, one thread begins running immediately, which is called the main thread. This thread is responsible for executing the main method of a program.

**Processes and Threads:**

* A process is an instance of a running program.
* A thread is a unit of execution within a process.
* Multiple threads can run concurrently within a single process.

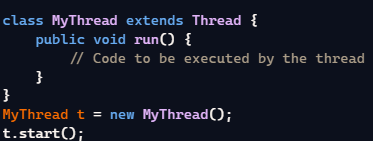
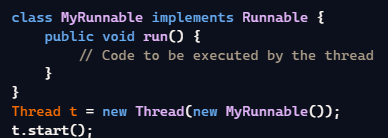
**Thread**  
A thread is the smallest unit of execution within a process. A process can have multiple threads that share resources but run independently.  
Threads can be created by extending the Thread class or implementing the Runnable interface

**Process vs. Thread:**  
A process is an independent program in execution, while threads are components of a process that can run concurrently

**Thread Lifecycle**

1. **New: Thread is created but not yet started.**
2. **Runnable: Thread is ready to run but waiting for CPU time.**
3. **Running: Thread is currently executing.**
4. **Blocked: Thread is waiting for a resource or event.**
5. **Terminated: Thread has finished execution**

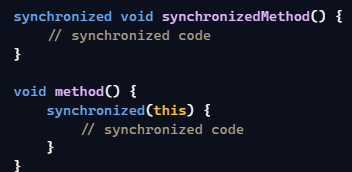
**Multitasking**  
Running multiple threads simultaneously to improve performance and resource utilization.  
Performing multiple tasks at the same time within a single program.  
Java supports multithreading through the Thread class and Runnable interface.

* Java supports multithreading through the Thread class and Runnable interface.  
   Benefits include improved application performance and responsiveness.

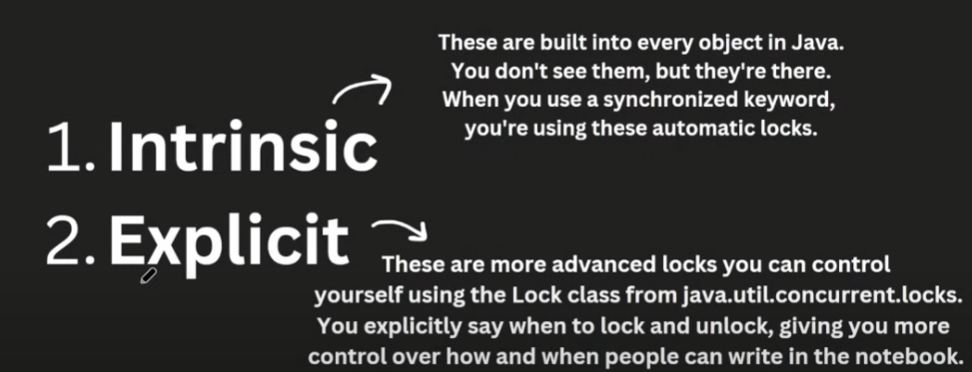
**Thread vs Runnable**  
Thread class: Suitable for simple applications.  
Runnable interface: Preferred for largescale applications as it allows separation of tasks from the execution mechanism.

**Methods**

* **start():** Begins the execution of the thread.
* **sleep(milliseconds):** Causes the thread to pause for a specified time 6.
* **join():** Waits for the thread to die 7.
* **yield(): S**uggests that the thread is willing to yield its current use of the CPU 8.
* **setPriority(int priority):** Sets the priority of the thread

**Synchronization**  
****

**Synchronization:** Ensures that only one thread can access a shared resource at a time.  
**synchronized keyword:** Used to synchronize blocks of code or entire methods.

**Locks**  
 ****

**ReentrantLock:**Provides more control over synchronization.

lock(), unlock(), and tryLock() methods.

**Yield**

Thread.yield() is a static method that suggests the current thread temporarily pause its execution to allow other threads of the same or higher priority to execute. It’s important to note that yield() is just a hint to the thread scheduler, and the actual behavior may vary depending on the JVM and OS.

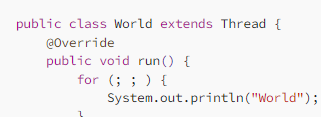
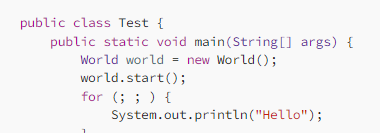
**Deadlock**

A deadlock occurs in concurrent programming when two or more threads are blocked forever, each waiting for the other to release a resource. This typically happens when threads hold locks on resources and request additional locks held by other threads. For example, Thread A holds Lock 1 and waits for Lock 2, while Thread B holds Lock 2 and waits for Lock 1. Since neither thread can proceed, they remain stuck in a deadlock state. Deadlocks can severely impact system performance and are challenging to debug and resolve in multi-threaded applications**.**

Interview Questions

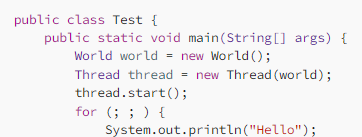
**Level 1: Beginner**

**1. create a new thread in Java?**

1. A new class World is created that extends Thread.
2. The run method is overridden to define the code that constitutes the new thread.
3. start method is called to initiate the new thread.  
   

**2. Implement Runnable interface**

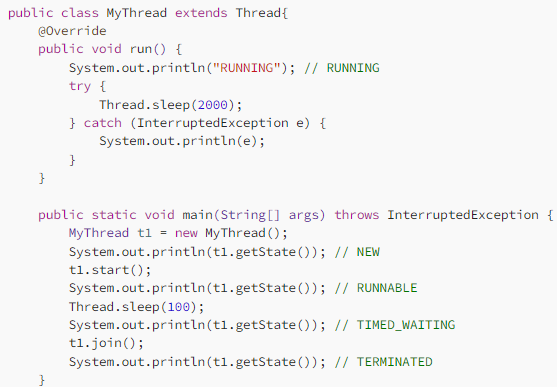
1. A new class World is created that implements Runnable.
2. The run method is overridden to define the code that constitutes the new thread.
3. A Thread object is created by passing an instance of World.
4. start method is called on the Thread object to initiate the new thread.



**3. Thread Lifecycle**

**The lifecycle of a thread in Java consists of several states, which a thread can move through during its execution.**

* **New**: A thread is in this state when it is created but not yet started.
* **Runnable**: After the start method is called, the thread becomes runnable. It’s ready to run and is waiting for CPU time.
* **Running**: The thread is in this state when it is executing.
* **Blocked/Waiting:** A thread is in this state when it is waiting for a resource or for another thread to perform an action.
* **Terminated**: A thread is in this state when it has finished executing.



**Core Concepts**

1. **What is a thread? How does it differ from a process?**

 A **thread** is a lightweight unit of execution within a process. Multiple threads can share the same memory space.

 A **process** is a separate instance of a program, with its own memory space.

1. **Explain the different states of a thread's lifecycle.**
2. **What is the difference between Runnable and Thread interfaces?**
3. **How can you create and start a thread in Java?**
4. **What is the difference between sleep() and wait() methods?**
5. **Explain the concept of thread priority.**
6. **What is the difference between yield() and join() methods?** **yield()** suggests that the current thread gives up the CPU to another thread of the same priority.
7. **join()** waits for the specified thread to finish before continuing.
8. **What is a daemon thread?**

A daemon thread is a background thread that doesn't prevent the JVM from exiting.

**Thread Synchronization**

1. **What is thread synchronization? Why is it necessary?**

Ensures that only one thread accesses a shared resource at a time to avoid data corruption.

1. **Explain the use of the synchronized keyword in Java.**
2. **What is a deadlock? How can it be avoided?**
3. **What is a race condition? How can it be prevented?**
4. **Explain the use of volatile keyword.**

Ensures that changes made to a variable by one thread are visible to other threads.

1. **What is the difference between synchronized and volatile keywords?**

**Concurrent Utilities**

1. **What is the java.util.concurrent package?**

Provides classes and interfaces for concurrent programming

1. **Explain the use of CountDownLatch and CyclicBarrier classes.**
2. **What is a Semaphore and how is it used?**
3. **Explain the use of ExecutorService and ThreadPoolExecutor classes.**
4. **What is the difference between Future and Callable interfaces?**
5. **Explain the use of CompletableFuture class.**Provides a flexible API for asynchronous programming and combining asynchronous operations.
6. **What is the ConcurrentHashMap class and how does it differ from HashMap?**A thread-safe concurrent hash map that offers better performance than synchronized HashMap.

**Advanced Topics**

1. **What is the difference between optimistic and pessimistic locking?**
2. **Explain the concept of thread-local storage.**
3. **What is the difference between blocking and non-blocking algorithms?**

 **Blocking:** Waits for an operation to complete before proceeding.

 **Non-blocking:** Attempts to perform an operation without blocking, and if it fails, retries or uses alternative approaches

1. **How can you measure the performance of a multithreaded application?**
2. **What are some common performance pitfalls in multithreaded programming?**

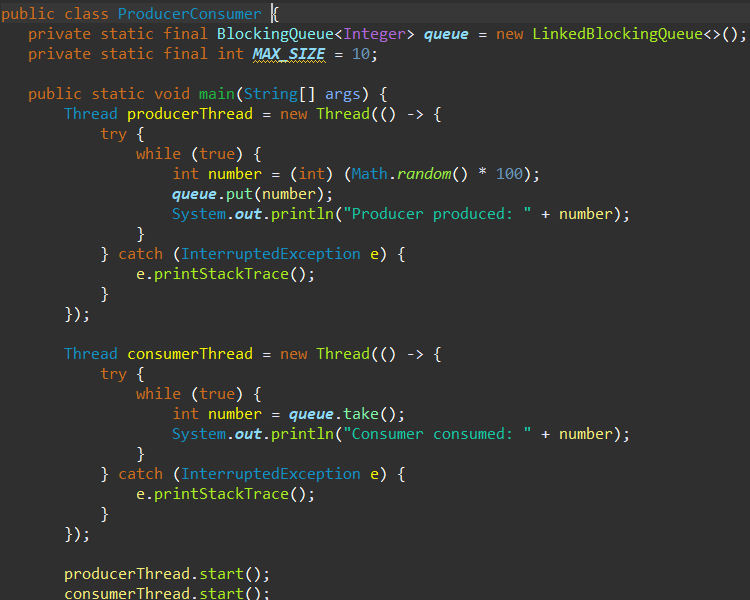
**Practical Scenarios**

1. **How would you design a multithreaded application to process a large number of tasks concurrently?**

 Use a thread pool to manage a fixed number of threads.

 Submit tasks to the pool for asynchronous execution

1. **How would you implement a producer-consumer pattern using Java's concurrent utilities?**

A classic concurrency pattern where one or more producer threads generate data and place it in a shared buffer, and one or more consumer threads consume data from the buffer.  


1. **How would you implement a thread-safe cache in Java?**
2. **How would you handle exceptions in a multithreaded environment?**
3. **How would you debug a multithreaded application**