Multithreading in JAVA

Top 20 Interview Questions with Answers

1. What is multithreading, and how is it different from multiprocessing?
2. Explain the lifecycle of a thread in Java.
3. What is the purpose of the Thread class and the Runnable interface? When would you use one over the other?
4. How do you create a thread in Java?
5. What is the difference between start()and run()methods in the Threadclass?
6. What is thread safety, and why is it important?
7. How can you achieve thread safety in Java?
8. Explain the concept of a synchronized block. How does it differ from a synchronized method?
9. What is the difference between a ReentrantLock and the synchronized keyword?
10. What is the purpose of the volatile keyword in Java? How is it different from synchronized?
11. What is a deadlock? How do you detect and prevent deadlocks in Java?
12. What is the difference between wait(), notify(), and notifyAll() methods in Java?
13. What is the purpose of the ThreadLocal class? When should it be used?
14. What is the difference between Callable and Runnable in Java? Why would you use Callable?
15. What are daemon threads in Java? How do you create a daemon thread?
16. What is the Fork/Join framework, and how is it different from traditional thread management?
17. How can you use the ExecutorService framework for managing threads?
18. What is the purpose of the Future and CompletableFuture classes in Java?
19. What are the benefits of using concurrent collections like ConcurrentHashMap or CopyOnWriteArrayList?
20. Explain how thread priorities work in Java. Can thread priorities guarantee execution order?

---

1. What is multithreading, and how is it different from multiprocessing?

Answer:
Multithreading allows concurrent execution of multiple threads within a single process, sharing memory and other resources.
Multiprocessing involves running multiple processes, each with its own memory space.

Example:

java

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread: " + Thread.currentThread().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        t1.start();
        t2.start();
    }
}
```

---

2. Explain the lifecycle of a thread in Java.

Answer:
Java threads go through the following states:

1. **NEW**: Thread is created but not started.
2. **RUNNABLE**: After calling start().
3. **BLOCKED**: Waiting for a lock.
4. **WAITING/TIMED_WAITING**: Waiting indefinitely or for a specified time.
5. **TERMINATED**: Thread execution is complete.

Example:

java

```
Thread t = new Thread(() -> System.out.println("Running thread..."));
System.out.println(t.getState()); // NEW
t.start();
System.out.println(t.getState()); // RUNNABLE
```

---

3. What is the purpose of the Thread class and the Runnable interface? When would you use one over the other?

Answer:

- Thread class: Use when the task does not require multiple inheritance.
- Runnable interface: Preferred for tasks where multiple inheritance is needed.

Example with Runnable:

java

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread running...");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start();
    }
}
```

---

4. How do you create a thread in Java?

Answer:
Threads can be created by:

1. Extending the Thread class.
2. Implementing the Runnable interface.

Example:

java

```
// Using Thread class
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running...");
    }
}

// Using Runnable interface
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread running...");
    }
}
```

---

5. What is the difference between start() and run() methods in the Thread class?

Answer:

- start(): Creates a new thread and calls the run() method.
- run(): Acts as a normal method call within the same thread.

Example:

java

```
class MyThread extends Thread {
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.run(); // Main thread
        t1.start(); // New thread
    }
}
```

---

6. What is thread safety, and why is it important?

Answer:
Thread safety ensures correct behavior when multiple threads access shared resources concurrently. Without it, race conditions and data corruption can occur.

---

7. How can you achieve thread safety in Java?

Answer:

1. **Synchronized blocks/methods.**
2. **Using concurrent collections** like ConcurrentHashMap.
3. **Using volatile for shared variables.**

Example:

java

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
```

```
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Thread t1 = new Thread(() -> counter.increment());
        Thread t2 = new Thread(() -> counter.increment());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Final count: " + counter.getCount());
    }
}
```

---

8. Explain the concept of a synchronized block. How does it differ from a synchronized method?

Answer:

- **Synchronized method** locks the entire method.
- **Synchronized block** locks only a part of the code.

Example:

java

```
class Counter {
    private int count = 0;

    public void increment() {
        synchronized (this) {  // Synchronized block
            count++;
        }
    }
}
```

---

9. What is the difference between a ReentrantLock and the synchronized keyword?

Answer:

- ReentrantLock provides more control (e.g., fairness policy, tryLock).
- synchronized is simpler to use but less flexible.

Example:

java

```
import java.util.concurrent.locks.ReentrantLock;

class Counter {
   private int count = 0;
   private final ReentrantLock lock = new ReentrantLock();

   public void increment() {
      lock.lock();
      try {
         count++;
      } finally {
         lock.unlock();
      }
   }
}
```

---

10. What is the purpose of the volatile keyword in Java?

Answer:
Ensures changes to a variable are visible to all threads and prevents thread-local caching.

Example:

java

```
class MyTask implements Runnable {
```

```
    private volatile boolean running = true;

    public void run() {
        while (running) {
            System.out.println("Thread is running...");
        }
    }

    public void stop() {
        running = false;
    }
}
```

---

11. What is a deadlock? How do you detect and prevent deadlocks in Java?

Answer:
Deadlocks occur when two threads wait for each other's locks.

Prevention:

1. Acquire locks in a fixed order.
2. Use tryLock with timeouts.

---

12. What is the difference between wait(), notify(), and notifyAll() methods?

Answer:

- wait(): Thread waits for a signal.
- notify(): Wakes up one thread.
- notifyAll(): Wakes up all threads.

Example:

java

```
class Shared {
    synchronized void waitForSignal() throws InterruptedException {
        wait();
```

```
    }

    synchronized void sendSignal() {
        notify();
    }
}
```

13. What is the purpose of the ThreadLocal class? When should it be used?

Answer:
The ThreadLocal class is used to create thread-local variables. Each thread accessing a ThreadLocal variable has its own independent copy.

Use case:
When you need data isolation for threads, like maintaining user sessions or storing transaction IDs.

Example:

java

```
class UserService {
    private static ThreadLocal<String> threadLocal = ThreadLocal.withInitial(() -> "Default User");

    public void setUser(String user) {
        threadLocal.set(user);
    }

    public String getUser() {
        return threadLocal.get();
    }
}

public class Main {
    public static void main(String[] args) {
        UserService userService = new UserService();

        Thread t1 = new Thread(() -> {
```

```
        userService.setUser("Alice");
        System.out.println("Thread 1 User: " + userService.getUser());
    });

    Thread t2 = new Thread(() -> {
        userService.setUser("Bob");
        System.out.println("Thread 2 User: " + userService.getUser());
    });

    t1.start();
    t2.start();
  }
}
```

---

14. What is the difference between Callable and Runnable in Java? Why would you use Callable?

Answer:

- **Runnable**: Does not return a result or throw checked exceptions.
- **Callable**: Can return a result and throw checked exceptions.

Example with Callable:

java

```
import java.util.concurrent.*;

class MyTask implements Callable<String> {
  public String call() throws Exception {
    return "Task completed!";
  }
}

public class Main {
  public static void main(String[] args) throws Exception {
```

```
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<String> future = executor.submit(new MyTask());
        System.out.println("Result: " + future.get());
        executor.shutdown();
    }
}
```

---

15. What are daemon threads in Java? How do you create a daemon thread?

Answer:
Daemon threads run in the background to support non-daemon threads. They terminate automatically when all non-daemon threads finish.

How to create: Use the setDaemon(true) method before starting the thread.

Example:

java

```
public class Main {
    public static void main(String[] args) {
        Thread daemonThread = new Thread(() -> {
            while (true) {
                System.out.println("Daemon thread running...");
            }
        });
        daemonThread.setDaemon(true);
        daemonThread.start();

        System.out.println("Main thread exiting...");
    }
}
```

---

16. What is the Fork/Join framework, and how is it different from traditional thread management?

Answer:
The Fork/Join framework, introduced in Java 7, uses the divide-and-conquer approach for parallel processing. It splits tasks into subtasks and merges the results.

Example:

java

```
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

class SumTask extends RecursiveTask<Integer> {
    private final int[] array;
    private final int start, end;

    public SumTask(int[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    protected Integer compute() {
        if (end - start <= 2) {
            int sum = 0;
            for (int i = start; i < end; i++) sum += array[i];
            return sum;
        } else {
            int mid = (start + end) / 2;
            SumTask leftTask = new SumTask(array, start, mid);
            SumTask rightTask = new SumTask(array, mid, end);
            leftTask.fork();
            return rightTask.compute() + leftTask.join();
        }
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool();
        int[] array = {1, 2, 3, 4, 5, 6};
        SumTask task = new SumTask(array, 0, array.length);
        int result = pool.invoke(task);
        System.out.println("Sum: " + result);
    }
}
```

---

17. How can you use the ExecutorService framework for managing threads?

Answer:
ExecutorService simplifies thread management by providing a thread pool. It efficiently handles thread creation and reusability.

Example:

java

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class Task implements Runnable {
    private final int taskId;

    public Task(int taskId) {
        this.taskId = taskId;
    }

    public void run() {
        System.out.println("Executing Task " + taskId + " by " + Thread.currentThread().getName());
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);
        for (int i = 1; i <= 5; i++) {
            executor.submit(new Task(i));
        }
        executor.shutdown();
    }
}
```

---

18. What is the purpose of the Future and CompletableFuture classes in Java?

Answer:

- **Future**: Represents the result of an asynchronous computation.
- **CompletableFuture**: Enhances Future by providing methods for chaining tasks and handling completion events.

Example with CompletableFuture:

java

```
import java.util.concurrent.CompletableFuture;

public class Main {
    public static void main(String[] args) {
        CompletableFuture.supplyAsync(() -> {
            System.out.println("Task started...");
            return 42;
        }).thenApply(result -> {
            System.out.println("Result: " + result);
            return result * 2;
        }).thenAccept(finalResult -> System.out.println("Final Result: " + finalResult));
```

```
    }
}
```

---

19. What are the benefits of using concurrent collections like ConcurrentHashMap or CopyOnWriteArrayList?

Answer:

- **Thread-safe without explicit synchronization**.
- Optimized for performance by reducing contention.
- Suitable for high-concurrency scenarios.

Example:

java

```
import java.util.concurrent.ConcurrentHashMap;

public class Main {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
        map.put("Key1", 1);
        map.put("Key2", 2);
        map.forEach((key, value) -> System.out.println(key + ": " + value));
    }
}
```

---

20. Explain how thread priorities work in Java. Can thread priorities guarantee execution order?

Answer:
Thread priorities (range 1 to 10) are hints to the thread scheduler. Higher-priority threads are favored but do not guarantee execution order due to platform dependency.

Example:

java

```
class Task extends Thread {
    public Task(String name) {
```

```
        super(name);
    }

    public void run() {
        System.out.println(Thread.currentThread().getName() + " with priority " +
Thread.currentThread().getPriority());
    }
}

public class Main {
    public static void main(String[] args) {
        Task t1 = new Task("Thread-1");
        Task t2 = new Task("Thread-2");
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.MAX_PRIORITY);
        t1.start();
        t2.start();
    }
}
```