Atomic Variables in Java

What are Atomic Variables in Java.Kindly explain it with example

What are Atomic Variables in Java?

Atomic variables in Java are part of the `java.util.concurrent.atomic` package, introduced in Java 1.5. They are designed to handle variables in a multithreaded environment without using locks. Atomic variables ensure **thread-safe operations** by utilizing low-level CPU instructions like Compare-And-Swap (CAS) for synchronization.

These variables provide methods to perform atomic updates, such as incrementing, decrementing, and comparing-and-setting values, without the need for synchronization blocks or `volatile` keywords.

Common Atomic Classes in Java

Some of the commonly used atomic classes include:

1. `AtomicInteger` - For integer values.
2. `AtomicLong` - For long values.
3. `AtomicBoolean` - For boolean values.
4. `AtomicReference<T>` - For object references.

Example: Using `AtomicInteger` in a Multithreaded Environment

```java
import java.util.concurrent.atomic.AtomicInteger;
```

```java
class Counter {
    private AtomicInteger count = new AtomicInteger(0);

    public void increment() {
        count.incrementAndGet(); // Atomically increments the value by 1
    }

    public int getValue() {
        return count.get(); // Atomically retrieves the current value
    }
}

public class AtomicExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        // Create two threads that increment the counter
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        });

        // Start the threads
        t1.start();
        t2.start();

        // Wait for threads to finish
        t1.join();
        t2.join();

        // Print the final counter value
        System.out.println("Final Count: " + counter.getValue());
    }
}
```

Explanation of the Code

1. **AtomicInteger Instance:**
    - The `` `AtomicInteger` `` object `` `count` `` is used instead of a regular `` `int` ``.

- All operations on this variable are atomic.

2. **Increment Operation:**
    - The `incrementAndGet()` method atomically increments the value by 1.
3. **Thread Safety:**
    - Even though multiple threads (`t1` and `t2`) update the counter concurrently, `AtomicInteger` ensures no race conditions occur.
4. **Expected Output:**
    - The final count will always be `2000`, even if the operations are executed in a highly concurrent environment.

Comparison with Synchronized Methods

Using `AtomicInteger`:

- No locks are required.
- Provides better performance than `synchronized` blocks.

Using `synchronized`:

- Introduces overhead due to thread locking and unlocking.
- Might be necessary when multiple operations need to be grouped into a critical section.

When to Use Atomic Variables

- **Fine-grained atomic updates**: Where operations like incrementing or setting a value need to be atomic.
- **High-performance applications**: Where using locks may create contention or slow down the application.

For complex scenarios requiring multiple operations to be atomic, consider using synchronized blocks or higher-level concurrency utilities like `ReentrantLock` or `StampedLock`.

what are AtomicInteger, AtomicLong, and AtomicBoolean.Kindly explain it with example

AtomicInteger, AtomicLong, and AtomicBoolean in Java

The `java.util.concurrent.atomic` package provides thread-safe classes for working with integer, long, and boolean values. These classes allow atomic operations on their values without the need for synchronization or explicit locking.

---

1. AtomicInteger

- Represents an integer value that can be updated atomically.
- Useful methods:
  - `get()`: Gets the current value.
  - `set(int newValue)`: Sets the value.
  - `incrementAndGet()`: Atomically increments by 1 and returns the new value.
  - `decrementAndGet()`: Atomically decrements by 1 and returns the new value.
  - `compareAndSet(expectedValue, newValue)`: Atomically sets the value to `newValue` if it matches `expectedValue`.

Example: AtomicInteger

```java
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicIntegerExample {
    public static void main(String[] args) throws InterruptedException {
        AtomicInteger counter = new AtomicInteger(0);

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.incrementAndGet(); // Atomically increment
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.incrementAndGet(); // Atomically increment
            }
        });

        t1.start();
```

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

        System.out.println("Final Counter Value: " + counter.get()); // Expected: 2000
    }
}
```

---

2. AtomicLong

- Similar to `` `AtomicInteger` ``, but for `` `long` `` values.
- Useful methods:
  - `` `get()` ``: Gets the current value.
  - `` `set(long newValue)` ``: Sets the value.
  - `` `addAndGet(long delta)` ``: Atomically adds a given value to the current value.
  - `` `getAndIncrement()` ``: Atomically increments and returns the previous value.

Example: AtomicLong

```java
import java.util.concurrent.atomic.AtomicLong;

public class AtomicLongExample {
    public static void main(String[] args) {
        AtomicLong totalSum = new AtomicLong(0);

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 500; i++) {
                totalSum.addAndGet(5); // Atomically add 5
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 500; i++) {
                totalSum.addAndGet(10); // Atomically add 10
            }
        });

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

```
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Total Sum: " + totalSum.get()); // Expected: 7500
    }
}
```

---

3. AtomicBoolean

- Represents a boolean value that can be updated atomically.
- Useful methods:
  - **`get()`**: Gets the current value.
  - **`set(boolean newValue)`**: Sets the value.
  - **`compareAndSet(expectedValue, newValue)`**: Sets the value if it matches the expected value.

Example: AtomicBoolean

```java
import java.util.concurrent.atomic.AtomicBoolean;

public class AtomicBooleanExample {
    public static void main(String[] args) {
        AtomicBoolean lock = new AtomicBoolean(false);

        Runnable task = () -> {
            if (lock.compareAndSet(false, true)) {
                System.out.println(Thread.currentThread().getName() + " acquired the lock!");
                // Perform some operation
                lock.set(false); // Release the lock
            } else {
                System.out.println(Thread.currentThread().getName() + " could not acquire the lock.");
            }
        };

        Thread t1 = new Thread(task, "Thread-1");
        Thread t2 = new Thread(task, "Thread-2");
```

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

Key Features and Use Cases

1. **AtomicInteger**
    - Counter variables (e.g., for tracking the number of tasks completed).
    - Alternative to `synchronized` for integer updates.
2. **AtomicLong**
    - Tracking cumulative values like sums or large counters.
    - Suitable for scenarios with long integers.
3. **AtomicBoolean**
    - Used for implementing lightweight locks or flags in multithreaded programs.

Advantages

- Thread-safe operations without using explicit locks or synchronization.
- Performance improvements due to the use of low-level CAS (Compare-And-Swap).

Limitations

- Suitable for simple atomic operations. For complex scenarios involving multiple variables, `synchronized` or `Lock` mechanisms are still necessary.

what are AtomicReference and Atomic ReferenceArray.Kindly explain it with example
 - what are Compare-and-Swap Operations.Kindly explain it with example

AtomicReference and AtomicReferenceArray

The `java.util.concurrent.atomic` package provides `AtomicReference` and `AtomicReferenceArray` for managing atomic operations on object references and arrays of references, respectively. These classes ensure thread-safe access to and modification of objects or array elements.

---

1. AtomicReference

- Represents an object reference that can be updated atomically.
- Provides methods to:
  - Get and set the reference atomically.
  - Compare and set the reference conditionally using **Compare-And-Swap (CAS)**.

Example: AtomicReference

```java
import java.util.concurrent.atomic.AtomicReference;

public class AtomicReferenceExample {
    public static void main(String[] args) {
        AtomicReference<String> atomicString = new AtomicReference<>("Hello");

        // Compare and set
        boolean wasUpdated = atomicString.compareAndSet("Hello", "World");
        System.out.println("Updated: " + wasUpdated + ", New Value: " + atomicString.get()); // Expected: true, World

        // Failed compare-and-set
        wasUpdated = atomicString.compareAndSet("Hello", "Atomic");
        System.out.println("Updated: " + wasUpdated + ", New Value: " + atomicString.get()); // Expected: false, World
    }
}
```

---

2. AtomicReferenceArray

- Represents an array of object references, where each element can be updated atomically.
- Useful for managing a collection of objects in a thread-safe way without using locks.

Example: AtomicReferenceArray

```java
import java.util.concurrent.atomic.AtomicReferenceArray;

public class AtomicReferenceArrayExample {
    public static void main(String[] args) {
        String[] initialArray = {"A", "B", "C"};
        AtomicReferenceArray<String> atomicArray = new AtomicReferenceArray<>(initialArray);

        // Update an element atomically
        boolean wasUpdated = atomicArray.compareAndSet(1, "B", "X");
        System.out.println("Updated: " + wasUpdated + ", New Value: " + atomicArray.get(1)); // Expected: true, X

        // Failed update
        wasUpdated = atomicArray.compareAndSet(2, "D", "Y");
        System.out.println("Updated: " + wasUpdated + ", New Value: " + atomicArray.get(2)); // Expected: false, C
    }
}
```

---

3. Compare-And-Swap (CAS) Operations

Compare-And-Swap (CAS) is a low-level atomic operation used to achieve thread-safe updates without locks. CAS works as follows:

1. **Compare**: Compares the current value of a variable with an expected value.
2. **Swap**: If they are equal, updates the variable to a new value.
3. **No Operation**: If they are not equal, no change is made.

Advantages of CAS:

- Avoids locking overhead.
- Ensures non-blocking operations, improving performance in highly concurrent environments.

How CAS Works (Conceptual Example)

```java
class CASExample {
    private volatile int value; // Shared value

    public CASExample(int initialValue) {
        this.value = initialValue;
    }

    public boolean compareAndSwap(int expectedValue, int newValue) {
        synchronized (this) {
            if (value == expectedValue) {
                value = newValue;
                return true;
            }
            return false;
        }
    }

    public int getValue() {
        return value;
    }
}

public class CompareAndSwapExample {
    public static void main(String[] args) {
        CASExample cas = new CASExample(10);

        boolean success = cas.compareAndSwap(10, 20);
        System.out.println("CAS Successful: " + success + ", New Value: " + cas.getValue()); // Expected: true, 20

        success = cas.compareAndSwap(10, 30);
        System.out.println("CAS Successful: " + success + ", New Value: " + cas.getValue()); // Expected: false, 20
    }
}
```

How CAS is Used in Atomic Classes

- **`compareAndSet(expectedValue, newValue)`** in classes like **`AtomicReference`** and **`AtomicInteger`** uses CAS internally.
- It retries the operation in case of contention until it succeeds.

Example: Using CAS with `AtomicInteger`

```java
import java.util.concurrent.atomic.AtomicInteger;

public class CASInAtomicInteger {
    public static void main(String[] args) {
        AtomicInteger atomicValue = new AtomicInteger(100);

        boolean updated = atomicValue.compareAndSet(100, 200);
        System.out.println("Updated: " + updated + ", New Value: " + atomicValue.get()); // Expected: true, 200

        updated = atomicValue.compareAndSet(100, 300);
        System.out.println("Updated: " + updated + ", New Value: " + atomicValue.get()); // Expected: false, 200
    }
}
```

Key Points

- **AtomicReference**: Thread-safe updates for single object references.
- **AtomicReferenceArray**: Thread-safe updates for arrays of object references.
- **CAS**: A low-level atomic operation used by these classes to ensure thread safety without locks.

These tools are useful for creating non-blocking, high-performance, and thread-safe programs in Java.

What are Lock and Semaphores in Java.

Locks and Semaphores in Java

In Java, **locks** and **semaphores** are synchronization mechanisms that help manage access to shared resources in multithreaded environments. Both are part of the `java.util.concurrent` package but serve different purposes.

---

1. Locks in Java

A **lock** is an advanced synchronization mechanism that allows one thread to control access to a shared resource. It provides more control over synchronization compared to the `` `synchronized` `` keyword.

Key Features of Locks:

- Allow multiple types of locks, such as **reentrant locks**.
- Can be **fair** or **non-fair** in terms of thread scheduling.
- Provide interruptible lock acquisition (`` `lockInterruptibly()` ``).
- Can try acquiring a lock with a timeout (`` `tryLock(timeout)` ``).

Lock Interface:

The `` `Lock` `` interface in Java defines methods like:

- `` `lock()` ``: Acquires the lock.
- `` `unlock()` ``: Releases the lock.
- `` `tryLock()` ``: Tries to acquire the lock without blocking indefinitely.

Common Implementations:

- **ReentrantLock**: A lock that allows the same thread to acquire it multiple times (reentrant behavior).
- **ReadWriteLock**: Allows multiple readers or one writer.

Example: Using ReentrantLock

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

public class LockExample {
    private final Lock lock = new ReentrantLock();
    private int counter = 0;

    public void increment() {
        lock.lock(); // Acquire the lock
        try {
```

```java
            counter++;
        } finally {
            lock.unlock(); // Always release the lock in the `finally` block
        }
    }

    public int getCounter() {
        return counter;
    }

    public static void main(String[] args) throws InterruptedException {
        LockExample example = new LockExample();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                example.increment();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                example.increment();
            }
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final Counter Value: " + example.getCounter()); // Expected: 2000
    }
}
```

---

2. Semaphores in Java

A **semaphore** is a synchronization tool that controls access to a shared resource by maintaining a count of available permits. It can be used to limit the number of threads accessing a resource simultaneously.

Types of Semaphores:

- **Counting Semaphore**: Allows a set number of threads to access a resource.
- **Binary Semaphore**: Works like a lock, allowing only one thread at a time.

Key Methods:

- `acquire()`: Acquires a permit, blocking if none are available.
- `release()`: Releases a permit, increasing the count.
- `tryAcquire()`: Attempts to acquire a permit without blocking.

Use Cases:

- Controlling access to a pool of resources (e.g., database connections, thread pools).
- Rate limiting or throttling.

Example: Using Semaphore to Limit Access

```java
import java.util.concurrent.Semaphore;

public class SemaphoreExample {
    private static final Semaphore semaphore = new Semaphore(3); // Allows 3 threads at a time

    public static void main(String[] args) {
        Runnable task = () -> {
            try {
                System.out.println(Thread.currentThread().getName() + " trying to acquire a permit.");
                semaphore.acquire(); // Acquire a permit
                System.out.println(Thread.currentThread().getName() + " acquired a permit.");

                // Simulate some work
                Thread.sleep(2000);

            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                System.out.println(Thread.currentThread().getName() + " releasing a permit.");
                semaphore.release(); // Release the permit
            }
        };

        // Create 5 threads, but only 3 can execute concurrently
        for (int i = 0; i < 5; i++) {
            new Thread(task, "Thread-" + (i + 1)).start();
        }
```

```
        }
    }
```

Output Explanation:

- Only 3 threads acquire permits and run concurrently.
- Once a thread releases a permit, the next waiting thread acquires it.

Comparison: Lock vs Semaphore

Feature	Lock	Semaphore
Purpose	Protects a shared resource for a single thread.	Controls access for multiple threads.
Type	Binary (lock/unlock).	Counting (permits).
Thread Access	One thread at a time (typically).	Multiple threads can acquire permits.
Interruptible	Yes, with methods like `` `lockInterruptibly()` ``.	Yes, with methods like `` `acquire()` ``.
Fairness	Can be fair or non-fair.	Can also enforce fairness.
Use Case	Thread-safe operations on a shared resource.	Managing resource pools, throttling, or rate limiting.

When to Use Which

- **Lock**: Use when you need fine-grained thread control over a single shared resource or critical section.
- **Semaphore**: Use when managing access to a pool of resources or limiting concurrent execution.

Explain below in details along with example:ReentrantLock
 - ReadWriteLock

ReentrantLock and ReadWriteLock in Java

Both `ReentrantLock` and `ReadWriteLock` are part of the `java.util.concurrent.locks` package and are used for more flexible and advanced thread synchronization than the traditional `synchronized` keyword.

---

1. ReentrantLock

A **ReentrantLock** is a lock implementation that allows the same thread to acquire the lock multiple times without causing a deadlock. Each `lock()` call must be matched with an `unlock()` call. It provides additional features compared to `synchronized`:

Key Features:

- **Reentrant**: A thread can acquire the lock multiple times without getting blocked.
- **Interruptible**: A thread waiting for a lock can be interrupted (`lockInterruptibly()`).
- **Fairness**: Supports fair (FIFO) ordering of threads waiting for the lock.
- **Try Locking**: You can attempt to acquire the lock without blocking (`tryLock()`).

Example: Using `ReentrantLock`

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

public class ReentrantLockExample {
    private final ReentrantLock lock = new ReentrantLock();
    private int counter = 0;

    public void increment() {
        lock.lock(); // Acquire the lock
        try {
            counter++;
            System.out.println(Thread.currentThread().getName() + " incremented counter to " + counter);
        } finally {
            lock.unlock(); // Always release the lock in a `finally` block
        }
    }

    public static void main(String[] args) {
        ReentrantLockExample example = new ReentrantLockExample();

        Runnable task = () -> {
            for (int i = 0; i < 3; i++) {
```

```
                example.increment();
            }
        };

        Thread t1 = new Thread(task, "Thread-1");
        Thread t2 = new Thread(task, "Thread-2");

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

Output Explanation:

- The `lock()` ensures that only one thread modifies the `counter` at a time.
- Threads take turns incrementing the counter, ensuring thread safety.

---

2. ReadWriteLock

A **ReadWriteLock** allows multiple threads to read a resource simultaneously but only one thread to write at a time. This is useful for scenarios where:

- Reads are frequent and do not need to block each other.
- Writes are infrequent but should block all other operations.

Key Components:

- **Read Lock**: Allows multiple threads to acquire the lock for reading if no thread has the write lock.
- **Write Lock**: Ensures exclusive access to the resource.

Common Implementation:

- **ReentrantReadWriteLock**: A concrete implementation of `ReadWriteLock`.

Example: Using `ReadWriteLock`

java

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

public class ReadWriteLockExample {
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private int value = 0;

    // Method for reading
    public int readValue() {
        lock.readLock().lock(); // Acquire the read lock
        try {
            System.out.println(Thread.currentThread().getName() + " read value: " + value);
            return value;
        } finally {
            lock.readLock().unlock(); // Release the read lock
        }
    }

    // Method for writing
    public void writeValue(int newValue) {
        lock.writeLock().lock(); // Acquire the write lock
        try {
            System.out.println(Thread.currentThread().getName() + " wrote value: " + newValue);
            value = newValue;
        } finally {
            lock.writeLock().unlock(); // Release the write lock
        }
    }

    public static void main(String[] args) {
        ReadWriteLockExample example = new ReadWriteLockExample();

        // Reading task
        Runnable readTask = () -> {
            for (int i = 0; i < 3; i++) {
                example.readValue();
            }
        };

        // Writing task
        Runnable writeTask = () -> {
            for (int i = 0; i < 3; i++) {
                example.writeValue(i);
            }
        };

        Thread t1 = new Thread(readTask, "Reader-1");
        Thread t2 = new Thread(readTask, "Reader-2");
        Thread t3 = new Thread(writeTask, "Writer");
```

```
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output Explanation:

- Multiple readers (`Reader-1` and `Reader-2`) can read the value concurrently.
- The writer (`Writer`) blocks all readers and other writers while it updates the value.

Comparison: ReentrantLock vs ReadWriteLock

Feature	ReentrantLock	ReadWriteLock
Purpose	Ensures mutual exclusion for a single thread.	Allows multiple readers or a single writer.
Concurrency	One thread at a time (for both read and write).	Multiple readers or one writer.
Use Case	Critical sections with both read and write access.	Scenarios with frequent reads and infrequent writes.
Fairness	Supports fairness if enabled.	Supports fairness if enabled.
API Simplicity	Simple `lock()` and `unlock()` methods.	Separate `readLock()` and `writeLock()` methods.

When to Use Which?

- Use **ReentrantLock** when:
  - You need mutual exclusion for all operations.
  - There's no distinction between read and write operations.
  - You require features like fairness or interruptibility.
- Use **ReadWriteLock** when:
  - There are frequent read operations that can be performed concurrently.
  - Write operations are less frequent but need exclusive access.

These locks are especially useful in scenarios where traditional `synchronized` blocks may not provide the desired flexibility or performance.

Explain below in details along with example:- StampedLock
- Semaphores
- Lock and Condition Interface

1. StampedLock

`StampedLock` is a lock introduced in Java 8 as part of the `java.util.concurrent.locks` package. It is designed for scenarios where high concurrency is required, and it supports three modes of locking:

- **Optimistic Read**: Lightweight, no blocking, suitable for read-heavy scenarios.
- **Read Lock**: Allows multiple readers but blocks writers.
- **Write Lock**: Exclusive access for a single writer.

Key Features:

- Optimized for read-heavy operations.
- Provides a stamp (long value) for lock verification and unlocking.
- Does not implement `ReentrantLock` behavior; a thread cannot reacquire the lock.

Example: Using StampedLock

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

public class StampedLockExample {
    private double x = 0, y = 0;
    private final StampedLock lock = new StampedLock();

    public void move(double deltaX, double deltaY) {
        long stamp = lock.writeLock(); // Acquire write lock
        try {
```

```java
            x += deltaX;
            y += deltaY;
        } finally {
            lock.unlockWrite(stamp); // Release write lock
        }
    }

    public double readDistanceFromOrigin() {
        long stamp = lock.tryOptimisticRead(); // Optimistic read lock
        double currentX = x, currentY = y;
        if (!lock.validate(stamp)) { // Check if no write occurred
            stamp = lock.readLock(); // Fallback to read lock
            try {
                currentX = x;
                currentY = y;
            } finally {
                lock.unlockRead(stamp);
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY);
    }

    public static void main(String[] args) {
        StampedLockExample example = new StampedLockExample();
        example.move(3, 4);
        System.out.println("Distance from origin: " + example.readDistanceFromOrigin());
    }
}
```

Output Explanation:

- The `move` method uses a write lock to update `x` and `y`.
- The `readDistanceFromOrigin` method uses an optimistic read lock for better performance, falling back to a regular read lock if necessary.

---

2. Semaphores

A **Semaphore** is a synchronization construct used to control access to a shared resource by multiple threads. It maintains a count of permits, which threads can acquire and release.

Types of Semaphores:

- **Counting Semaphore**: Allows multiple threads to access a resource.

- **Binary Semaphore**: Works like a lock with only one permit.

Example: Using Semaphore to Limit Access

```java
import java.util.concurrent.Semaphore;

public class SemaphoreExample {
    private static final Semaphore semaphore = new Semaphore(3); // 3 permits

    public static void main(String[] args) {
        Runnable task = () -> {
            try {
                System.out.println(Thread.currentThread().getName() + " trying to acquire a permit.");
                semaphore.acquire(); // Acquire a permit
                System.out.println(Thread.currentThread().getName() + " acquired a permit.");

                // Simulate some work
                Thread.sleep(2000);

            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                System.out.println(Thread.currentThread().getName() + " releasing a permit.");
                semaphore.release(); // Release the permit
            }
        };

        // Start 5 threads; only 3 can execute concurrently
        for (int i = 0; i < 5; i++) {
            new Thread(task, "Thread-" + (i + 1)).start();
        }
    }
}
```

Output Explanation:

- At most 3 threads can acquire the semaphore and execute concurrently.
- Other threads wait until permits are released.

---

3. Lock and Condition Interface

Lock Interface

The `Lock` interface provides thread synchronization mechanisms with more capabilities than `synchronized`. It allows:

- Fine-grained control over locking.
- Interruptible lock acquisition.
- Try-lock operations with timeout.

Example: Using Lock Interface

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

public class LockExample {
    private final Lock lock = new ReentrantLock();
    private int counter = 0;

    public void increment() {
        lock.lock();
        try {
            counter++;
            System.out.println(Thread.currentThread().getName() + " incremented counter to " + counter);
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        LockExample example = new LockExample();
        Runnable task = example::increment;

        Thread t1 = new Thread(task, "Thread-1");
        Thread t2 = new Thread(task, "Thread-2");

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

Condition Interface

The `Condition` interface works with the `Lock` interface to allow threads to wait and be signaled. It provides finer control over thread communication than `Object.wait()` and `Object.notify()`.

Key Methods:

- `await()`: Causes the current thread to wait until signaled.
- `signal()`: Wakes up one waiting thread.
- `signalAll()`: Wakes up all waiting threads.

Example: Using Lock and Condition

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

public class ConditionExample {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private boolean ready = false;

    public void producer() throws InterruptedException {
        lock.lock();
        try {
            System.out.println("Producer producing...");
            ready = true;
            condition.signal(); // Signal the consumer
        } finally {
            lock.unlock();
        }
    }

    public void consumer() throws InterruptedException {
        lock.lock();
        try {
            while (!ready) {
                System.out.println("Consumer waiting...");
                condition.await(); // Wait until signaled
            }
            System.out.println("Consumer consuming...");
        } finally {
            lock.unlock();
        }
    }
```

```
    public static void main(String[] args) {
        ConditionExample example = new ConditionExample();

        Thread producerThread = new Thread(() -> {
            try {
                Thread.sleep(1000);
                example.producer();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        Thread consumerThread = new Thread(() -> {
            try {
                example.consumer();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        consumerThread.start();
        producerThread.start();
    }
}
```

Output Explanation:

- The `consumer` thread waits for the `producer` thread to signal that it's ready.
- The `producer` thread signals the condition after setting `ready` to `true`.

Comparison: Lock, StampedLock, Semaphore, and Condition

Feature	Lock	StampedLock	Semaphore	Condition
Purpose	General synchronization	Optimized for read-heavy	Limit concurrent access	Thread communication
Fairness	Optional	No	Optional	Not applicable
Reentrant	Yes	No	No	N/A
Optimistic Read	No	Yes	No	N/A
Use Case	Critical sections	Read-heavy operations	Resource pools	Producer-consumer

Each tool has its strengths and is suited for specific concurrency challenges.

What are Thread Safety Best Practices

Thread Safety Best Practices

Thread safety refers to ensuring that shared data or resources are accessed and modified in a way that prevents race conditions and ensures predictable behavior in a multithreaded environment. Below are some best practices to achieve thread safety in Java:

---

1. Use Immutable Objects

- Immutable objects cannot be modified after creation, making them inherently thread-safe.
- Use classes like `String`, `Integer`, or custom immutable objects.

Example:

```java
public final class ImmutableData {
    private final int value;

    public ImmutableData(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

---

2. Use `synchronized` for Critical Sections

- Synchronize access to shared resources to ensure only one thread can execute a critical section at a time.

Example:

```java
java


public class SynchronizedExample {
    private int counter = 0;

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

    public synchronized int getCounter() {
        return counter;
    }
}
```

---

3. Use `java.util.concurrent` Utilities

- Prefer classes and tools from the `java.util.concurrent` package for thread-safe operations.
- Examples:
  - **Thread-safe collections**: `ConcurrentHashMap`, `CopyOnWriteArrayList`.
  - **Executor Framework**: `ExecutorService` for managing thread pools.
  - **Locks**: `ReentrantLock` or `ReadWriteLock` for finer control.

Example: Concurrent Collection

```java
java


import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentExample {
    private final ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

    public void add(String key, int value) {
        map.put(key, value);
    }

    public int get(String key) {
        return map.get(key);
```

```
    }
}
```

4. Use Atomic Variables

- Use classes like `` `AtomicInteger` ``, `` `AtomicLong` ``, and `` `AtomicReference` `` for atomic updates on variables without using locks.

Example:

```java
import java.util.concurrent.atomic.AtomicInteger;

public class AtomicExample {
    private final AtomicInteger counter = new AtomicInteger(0);

    public void increment() {
        counter.incrementAndGet();
    }

    public int getCounter() {
        return counter.get();
    }
}
```

5. Use Final or Volatile for Shared Variables

- Use `` `final` `` for variables that don't change after initialization.
- Use `` `volatile` `` for variables that are shared across threads and updated frequently to ensure visibility.

Example:

```java
public class VolatileExample {
    private volatile boolean flag = false;
```

```
    public void setFlag(boolean value) {
        flag = value;
    }

    public boolean getFlag() {
        return flag;
    }
}
```

---

6. Avoid Shared State or Use Thread-Local Storage

- Avoid sharing mutable state between threads where possible.
- Use `ThreadLocal` for variables that should have a separate copy for each thread.

Example:

```java
public class ThreadLocalExample {
    private static final ThreadLocal<Integer> threadLocalValue = ThreadLocal.withInitial(() -> 0);

    public void set(int value) {
        threadLocalValue.set(value);
    }

    public int get() {
        return threadLocalValue.get();
    }
}
```

---

7. Minimize Lock Scope

- Keep the scope of locks as small as possible to reduce contention and improve performance.

Example:

```java
public class MinimizedLockScope {
    private final Object lock = new Object();
    private int counter = 0;

    public void increment() {
        synchronized (lock) {
            counter++;
        }
    }
}
```

---

8. Use Stateless Objects

- Stateless objects, which don't maintain any state, are inherently thread-safe.
- Example: Utility classes like `Math`.

---

9. Avoid Deadlocks

- Deadlocks occur when multiple threads are waiting for each other to release resources.
- To prevent deadlocks:
  - Acquire locks in a consistent order.
  - Use `tryLock()` with a timeout.

Example:

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

public class DeadlockAvoidance {
    private final ReentrantLock lock1 = new ReentrantLock();
    private final ReentrantLock lock2 = new ReentrantLock();

    public void task1() {
        if (lock1.tryLock()) {
```

```
            try {
                if (lock2.tryLock()) {
                    try {
                        System.out.println("Task 1 completed");
                    } finally {
                        lock2.unlock();
                    }
                }
            } finally {
                lock1.unlock();
            }
        }
    }
}
```

---

10. Use Thread-safe Design Patterns

- Apply thread-safe patterns like:
  - **Producer-Consumer**: Use `BlockingQueue` for safe data exchange.
  - **Singleton**: Use `Enum` or double-checked locking for thread-safe singletons.

Example: Thread-safe Singleton

```java
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

11. Prefer Executors over Threads

- Use the `Executor` framework to manage thread pools instead of creating threads manually.

Example:

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

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        Runnable task = () -> System.out.println("Task executed by: " + Thread.currentThread().getName());

        for (int i = 0; i < 5; i++) {
            executor.submit(task);
        }

        executor.shutdown();
    }
}
```

---

12. Avoid Long Locks

- Avoid holding locks for long periods. Break tasks into smaller units if possible.

---

13. Test Thread Safety

- Test code for thread safety using tools like:
  - **Multithreaded tests**: Simulate concurrent execution.
  - Tools like **FindBugs** or **SonarQube** to detect concurrency issues.

---

14. Use Libraries for Concurrency

- Leverage libraries like:
  - Akka (Actor model for concurrency).
  - RxJava (Reactive programming).

Summary

To achieve thread safety:

1. Use immutable objects, atomic classes, and thread-safe collections.
2. Synchronize access to shared resources carefully.
3. Use advanced concurrency utilities like `Locks`, `Executors`, and `BlockingQueues`.
4. Avoid common pitfalls like deadlocks and race conditions.
5. Keep locks short and prefer stateless or thread-local designs.

Explain the concept of Immutable Objects
 - ThreadLocal Usage.How do we create Custom Immutable objects in JAva

Immutable Objects in Java

An **immutable object** is an object whose state cannot be modified after it is created. These objects are inherently thread-safe since their state cannot change, eliminating the risk of race conditions.

Characteristics of Immutable Objects

1. **State cannot change after creation.**
2. **All fields are `final`.**
3. **No setters or mutators.**
4. **The `this` reference does not escape during object construction.**

5. **Any mutable objects passed to the constructor are deeply copied.**

Examples of Immutable Objects

1. Built-in immutable classes in Java: `String`, `Integer`, `Double`, etc.
2. Custom immutable objects.

Creating a Custom Immutable Object

Follow these steps to create a custom immutable object:

1. **Declare the class as `final`** to prevent subclassing.
2. **Make all fields `private` and `final`.**
3. **Do not provide setters.**
4. **Initialize fields through a constructor only.**
5. **Ensure that any mutable object passed to the constructor or returned by a getter is cloned or deeply copied.**

Example: Custom Immutable Object

```java
import java.util.Date;

public final class ImmutableObject {
    private final String name;
    private final int age;
    private final Date dateOfBirth;

    public ImmutableObject(String name, int age, Date dateOfBirth) {
        this.name = name;
        this.age = age;
        // Create a defensive copy of the mutable object
        this.dateOfBirth = new Date(dateOfBirth.getTime());
    }

    public String getName() {
        return name;
    }
```

```
    public int getAge() {
        return age;
    }

    public Date getDateOfBirth() {
        // Return a defensive copy of the mutable object
        return new Date(dateOfBirth.getTime());
    }
}
```

Usage:

```java
import java.util.Date;

public class Main {
    public static void main(String[] args) {
        Date dob = new Date();
        ImmutableObject obj = new ImmutableObject("John", 30, dob);

        System.out.println("Name: " + obj.getName());
        System.out.println("Age: " + obj.getAge());
        System.out.println("DOB: " + obj.getDateOfBirth());

        // Attempt to modify the date externally
        dob.setTime(0);
        System.out.println("Modified DOB (outside): " + obj.getDateOfBirth()); // Original DOB remains unchanged
    }
}
```

Output:

- The object's internal state remains unmodified, ensuring immutability.

ThreadLocal in Java

The `ThreadLocal` class in Java provides thread-local variables. Each thread accessing such a variable gets its own, independently initialized copy, ensuring no interference between threads.

Key Methods of `ThreadLocal`

1. `get()`: Retrieves the value for the current thread.
2. `set(T value)`: Sets the value for the current thread.
3. `remove()`: Removes the value associated with the current thread.
4. `initialValue()`: Provides the initial value for the thread-local variable (can be overridden).

ThreadLocal Usage

Example: Using `ThreadLocal` to Maintain Thread-Specific Data

```java
public class ThreadLocalExample {
    private static final ThreadLocal<Integer> threadLocalValue = ThreadLocal.withInitial(() -> 1);

    public static void main(String[] args) {
        Runnable task = () -> {
            Integer value = threadLocalValue.get();
            System.out.println(Thread.currentThread().getName() + " initial value: " + value);

            threadLocalValue.set(value + 1);
            System.out.println(Thread.currentThread().getName() + " updated value: " + threadLocalValue.get());
        };

        Thread thread1 = new Thread(task, "Thread-1");
        Thread thread2 = new Thread(task, "Thread-2");

        thread1.start();
        thread2.start();
    }
}
```

Output:

- Each thread gets its own independent copy of the variable.
- No interference occurs between `Thread-1` and `Thread-2`.

ThreadLocal with Custom Objects

Example: Using ThreadLocal to Store User Information

```java
public class UserContext {
    private static final ThreadLocal<String> currentUser = new ThreadLocal<>();

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

    public static String getUser() {
        return currentUser.get();
    }

    public static void clear() {
        currentUser.remove();
    }
}

public class ThreadLocalUserExample {
    public static void main(String[] args) {
        Runnable task1 = () -> {
            UserContext.setUser("Alice");
            System.out.println(Thread.currentThread().getName() + " user: " + UserContext.getUser());
            UserContext.clear();
        };

        Runnable task2 = () -> {
            UserContext.setUser("Bob");
            System.out.println(Thread.currentThread().getName() + " user: " + UserContext.getUser());
            UserContext.clear();
        };

        Thread t1 = new Thread(task1, "Thread-1");
        Thread t2 = new Thread(task2, "Thread-2");

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

Output:

- Each thread independently manages its user information.

Advantages of ThreadLocal

1. **Thread Safety**: No need for synchronization.
2. **Independent Thread Data**: Each thread gets its own copy of the variable.
3. **Encapsulation**: Useful for storing thread-specific information like user IDs, transaction details, etc.

Key Differences: Immutable Objects vs ThreadLocal

Feature	Immutable Objects	ThreadLocal
Definition	Objects whose state cannot be modified.	Thread-specific storage for variables.
Thread Safety	Achieved through immutability.	Achieved through thread-local isolation.
Use Case	Shared data that doesn't change.	Thread-specific data like user sessions.
Complexity	Simple to implement.	Requires `` `ThreadLocal` `` API usage.

By combining **immutable objects** and **ThreadLocal**, Java provides powerful tools to manage thread safety and state isolation effectively.

Explain below points in details-Double-Checked Locking and its Issues
 - Concurrency Design Patterns

Double-Checked Locking and Its Issues

What is Double-Checked Locking?

Double-Checked Locking is an optimization technique to reduce the overhead of acquiring a lock by ensuring that the critical section is accessed only once after the initialization has been completed. It is commonly used for implementing thread-safe singleton patterns.

Steps in Double-Checked Locking

1. Check if the instance is already initialized **without acquiring the lock**.
2. If uninitialized, acquire the lock and check again inside the synchronized block (to ensure no other thread initialized it in the meantime).
3. Initialize the object if it is still uninitialized.

Example: Double-Checked Locking Singleton

```java
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {
        // Private constructor
    }

    public static Singleton getInstance() {
        if (instance == null) { // First check (no locking)
            synchronized (Singleton.class) {
                if (instance == null) { // Second check (with locking)
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Why Use `volatile`?

The `volatile` keyword ensures proper visibility and ordering of the instance variable across threads. Without `volatile`, threads might see a partially constructed object due to instruction reordering during object initialization.

Issues with Double-Checked Locking (Pre-Java 1.5)

1. **Instruction Reordering**:

- The Java Memory Model (JMM) before Java 1.5 allowed the following reordering during instance creation:
  - Allocate memory.
  - Assign memory to `instance` (making it non-null).
  - Initialize the object.
- A thread might see a non-null `instance` before it is fully initialized, leading to potential errors.

2. **Complexity**:
   - Double-checked locking introduces complexity and is prone to subtle bugs if not implemented correctly.
3. **Solution**:
   - Starting from Java 1.5, the `volatile` keyword solves the reordering issue.

Alternatives to Double-Checked Locking

1. **Eager Initialization**:
   - The instance is created at class loading time.

```java
public class EagerSingleton {
    private static final EagerSingleton INSTANCE = new EagerSingleton();

    private EagerSingleton() {}

    public static EagerSingleton getInstance() {
        return INSTANCE;
    }
}
```

2. **Bill Pugh Singleton (Holder Pattern)**:
   - A lazy-loaded and thread-safe approach without synchronization overhead.

```java
public class Singleton {
    private Singleton() {}
```

```
    private static class Holder {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return Holder.INSTANCE;
    }
}
```

Concurrency Design Patterns

Concurrency design patterns provide reusable solutions to common problems encountered in multithreaded and parallel programming. These patterns help manage shared resources, improve performance, and ensure thread safety.

---

1. Singleton Pattern

Ensures that only one instance of a class is created and provides a global access point to it.

- Use lazy initialization or double-checked locking for thread safety.

---

2. Producer-Consumer Pattern

Separates tasks into **producers** (that generate data) and **consumers** (that process data). A shared queue is used to exchange data between them.

Example:

```java
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class ProducerConsumerExample {
    private static final BlockingQueue<Integer> queue = new LinkedBlockingQueue<>();

    public static void main(String[] args) {
        Thread producer = new Thread(() -> {
            try {
```

```
                for (int i = 0; i < 5; i++) {
                    System.out.println("Produced: " + i);
                    queue.put(i);
                    Thread.sleep(100);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        Thread consumer = new Thread(() -> {
            try {
                while (true) {
                    int value = queue.take();
                    System.out.println("Consumed: " + value);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        producer.start();
        consumer.start();
    }
}
```

---

3. Thread Pool Pattern

A fixed number of threads are reused to execute tasks. This reduces the overhead of creating and destroying threads.

Example:

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

public class ThreadPoolExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 5; i++) {
            int task = i;
```

```
            executor.execute(() -> System.out.println("Executing Task: " + task + " by " + Thread.currentThread().getName()));
        }

        executor.shutdown();
    }
}
```

---

4. Read-Write Lock Pattern

Allows multiple threads to read shared data simultaneously while ensuring exclusive access for write operations.

Example:

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

public class ReadWriteLockExample {
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private int value = 0;

    public int read() {
        lock.readLock().lock();
        try {
            return value;
        } finally {
            lock.readLock().unlock();
        }
    }

    public void write(int newValue) {
        lock.writeLock().lock();
        try {
            value = newValue;
        } finally {
            lock.writeLock().unlock();
        }
    }
}
```

---

5. Future Pattern

Used to represent the result of an asynchronous computation.

Example:

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

public class FutureExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Callable<String> task = () -> {
            Thread.sleep(1000);
            return "Task completed";
        };

        Future<String> future = executor.submit(task);

        System.out.println("Task submitted, doing other work...");
        System.out.println("Result: " + future.get());

        executor.shutdown();
    }
}
```

---

6. Fork/Join Pattern

Efficiently splits tasks into smaller subtasks, executes them in parallel, and combines the results. Commonly used in divide-and-conquer algorithms.

Example:

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

public class ForkJoinExample extends RecursiveTask<Integer> {
    private final int start, end;

    public ForkJoinExample(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    protected Integer compute() {
        if (end - start <= 10) {
            int sum = 0;
            for (int i = start; i <= end; i++) {
                sum += i;
            }
            return sum;
        }

        int mid = (start + end) / 2;
        ForkJoinExample leftTask = new ForkJoinExample(start, mid);
        ForkJoinExample rightTask = new ForkJoinExample(mid + 1, end);

        leftTask.fork();
        int rightResult = rightTask.compute();
        int leftResult = leftTask.join();

        return leftResult + rightResult;
    }

    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool();
        int result = pool.invoke(new ForkJoinExample(1, 100));
        System.out.println("Sum: " + result);
    }
}
```

Summary of Concurrency Design Patterns

Pattern	Description	Use Case
Singleton	Ensures only one instance exists.	Global configuration objects.
Producer-Consumer	Separates producers from consumers using a shared queue.	Data pipelines, logging.
Thread Pool	Reuses a fixed number of threads.	Task execution, web servers.
Read-Write Lock	Allows multiple readers but one writer.	Read-heavy shared data.
Future	Represents the result of an asynchronous computation.	Asynchronous programming, background tasks.
Fork/Join	Efficient parallel execution of recursive tasks.	Divide-and-conquer algorithms.

Each pattern provides a specific solution to common concurrency challenges, ensuring better performance and maintainability in multithreaded applications.