Step-By-Step guide that covers topics from beginner to advanced levels:

Beginner Level:

1. Understanding the Basics:
   - Start by grasping the concept of multithreading and why it's important.
   - Learn about the `Thread` class in Java and how to create and run threads.

2. Synchronization:
   - Study how to use the `synchronized` keyword to prevent data corruption in multi-threaded programs.
   - Understand the concept of locks and critical sections.

3. Thread States:
   - Learn about different thread states like NEW, RUNNABLE, BLOCKED, WAITING, and TERMINATED.
   - Understand how threads transition between states.

4. Thread Priorities:
   - Explore thread priorities to control the execution order of threads.
   - Understand the `setPriority()` method.

5. Daemon Threads:
   - Learn about daemon threads and how they differ from non-daemon threads.
   - Use `setDaemon()` to set a thread as a daemon.

Intermediate Level:

6. Thread Pools:
   - Study the concept of thread pools and how to create them using `ExecutorService`.
   - Understand the benefits of thread pooling.

7. Callable and Future:
   - Learn about the `Callable` interface and the `Future` class for handling thread results.
   - Explore how to retrieve results from threads.

8. Synchronization Mechanisms:
   - Dive deeper into synchronization with mechanisms like `wait()`, `notify()`, and `notifyAll()`.
   - Understand the `Lock` and `Condition` interfaces.

9. Atomic Variables:
   - Explore atomic classes like `AtomicInteger` and `AtomicLong` for thread-safe operations.
   - Understand the benefits of atomic variables over traditional synchronization.

Advanced Level:

10. Thread Communication:

- Study advanced thread communication techniques using `CountDownLatch`, `CyclicBarrier`, and `Semaphore`.
- Learn when and how to use each of these constructs.

11. Thread Safety and Concurrency:
- Dive deep into concurrent data structures such as `ConcurrentHashMap`, `CopyOnWriteArrayList`, and `BlockingQueue`.
- Understand when and how to use them effectively.

12. Fork/Join Framework:
- Learn about the fork/join framework for parallel processing of tasks.
- Understand how to divide and conquer problems using this framework.

13. Thread Interference and Memory Consistency:
- Explore the Java Memory Model (JMM) and how it affects thread interaction.
- Learn about volatile keyword and happens-before relationship.

14. Java 8+ Features:
- Study the concurrency enhancements in Java 8 and later, such as `CompletableFuture`, the `Stream` API, and parallel streams.
- Learn how to leverage these features for concurrent programming.

15. Advanced Concepts:
- Study more advanced topics like thread profiling, deadlock detection, and thread dumps.
- Learn how to diagnose and resolve complex concurrency issues.

16. Concurrency Patterns:
- Explore common concurrency patterns like the Singleton pattern, the Producer-Consumer pattern, and the Reader-Writer pattern.
- Implement these patterns in your multithreaded applications.

Detailed :

Java Multithreading:
Multithreading is a technique in Java (and other programming languages) where a program can have multiple threads running concurrently. A thread is a lightweight sub-process within a program, and multithreading allows you to execute multiple parts of your program simultaneously, making better use of CPU resources and improving performance.

A simple example of how to create two threads in Java **to print odd and even numbers alternatively:**

```
public class OddEvenPrinter {
    public static void main(String[] args) {
```

```
        // Create two instances of Runnable for odd and even numbers
        Runnable oddPrinter = new OddRunnable();
        Runnable evenPrinter = new EvenRunnable();

        // Create two threads for each Runnable
        Thread oddThread = new Thread(oddPrinter);
        Thread evenThread = new Thread(evenPrinter);

        // Start both threads
        oddThread.start();
        evenThread.start();
    }
}

class OddRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i <= 10; i += 2) {
            System.out.println("Odd: " + i);
            try {
                Thread.sleep(100); // Adding a small delay for better
visualization
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class EvenRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 2; i <= 10; i += 2) {
            System.out.println("Even: " + i);
            try {
                Thread.sleep(100); // Adding a small delay for better
visualization
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

In this code:

1. We create a `OddEvenPrinter` class with a `main` method to kickstart our program.
2. We create two separate `Runnable` instances, `OddRunnable` and `EvenRunnable`, to encapsulate the logic for printing odd and even numbers, respectively.
3. Two threads, `oddThread` and `evenThread`, are created, each associated with one of the `Runnable` instances.
4. We start both threads using the `start` method.
5. The `OddRunnable` prints odd numbers (1, 3, 5, 7, 9), and the `EvenRunnable` prints even numbers (2, 4, 6, 8, 10). We add a small delay between each print for better visualization.

When you run this code, you'll see odd and even numbers printed by two separate threads, and the output may not be in a strict sequential order due to the concurrent execution of threads.

Another Example :

Example Code:
Let's create a simple Java program that demonstrates multithreading by calculating the sum of numbers in two threads concurrently. We will create two threads, each calculating the sum of numbers in a separate range.

```
public class MultithreadingExample {
    public static void main(String[] args) {
        // Define the range for the first thread
        int start1 = 1;
        int end1 = 5000;

        // Define the range for the second thread
        int start2 = 5001;
        int end2 = 10000;

        // Create two thread instances
        Thread thread1 = new Thread(new SumCalculator(start1, end1));
        Thread thread2 = new Thread(new SumCalculator(start2, end2));

        // Start the threads
        thread1.start();
        thread2.start();

        try {
            // Wait for both threads to finish
            thread1.join();
            thread2.join();
```

```
            // Calculate the total sum
            long totalSum = SumCalculator.sum1 + SumCalculator.sum2;
            System.out.println("Total Sum: " + totalSum);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class SumCalculator implements Runnable {
    private int start;
    private int end;
    public static long sum1 = 0;
    public static long sum2 = 0;

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

    @Override
    public void run() {
        long sum = 0;
        for (int i = start; i <= end; i++) {
            sum += i;
        }

        if (start == 1) {
            sum1 = sum;
        } else {
            sum2 = sum;
        }
    }
}
```

Explanation:
In this code:

1. We create a `MultithreadingExample` class with the `main` method. This is the entry point of our program.

2. We define two ranges for numbers to be summed by two threads: `start1` to `end1` and `start2` to `end2`.
3. Two `Thread` instances (`thread1` and `thread2`) are created, each associated with a `SumCalculator` object, which implements the `Runnable` interface.

4. We start both threads using the `start` method.

5. We use the `join` method to wait for both threads to finish their work.

6. Once both threads have completed their calculations, we add the results and print the total sum.

Synchronization in Multithreading :

Synchronization in multithreaded programming is required to ensure data consistency and avoid race conditions. Race conditions occur when multiple threads access shared data simultaneously, leading to unpredictable and erroneous results. Synchronization is necessary for the following reasons:

1. Atomicity: Some operations are not inherently atomic, meaning they can be interrupted by other threads. For example, reading and writing to a variable in multiple steps can be interrupted, causing inconsistencies. Synchronization ensures that such operations are completed without interruption.

2. Visibility: In a multithreaded environment, threads can cache variables locally, making changes in one thread invisible to others. Synchronization mechanisms guarantee that changes made by one thread become visible to other threads, preventing stale or incorrect data.

3. Consistency: Synchronization ensures that the program maintains a consistent state, even when multiple threads are modifying shared data. It allows threads to coordinate their actions to achieve a predictable outcome.

4. Preventing Deadlocks: Synchronization mechanisms also help in preventing deadlocks, which occur when two or more threads are unable to proceed because each is waiting for a resource held by the other. Proper synchronization can help avoid such situations.

Example Code:

Here's a simple example to illustrate the need for synchronization and how it can prevent data corruption:

```
public class SynchronizationExample {
    private static int sharedCounter = 0;

    public static void main(String[] args) {
        // Create two threads that increment a shared counter
        Thread thread1 = new Thread(new IncrementTask());
```

```
        Thread thread2 = new Thread(new IncrementTask());

        thread1.start();
        thread2.start();

        try {
            thread1.join();
            thread2.join();

            System.out.println("Final Shared Counter: " +
sharedCounter);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    static class IncrementTask implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++) {
                sharedCounter++; // Increment the shared counter
            }
        }
    }
}
```

In this code, we have two threads (`thread1` and `thread2`) running concurrently, both incrementing a shared variable `sharedCounter`. If we run this code without synchronization, you'll likely observe inconsistent or incorrect results because both threads are modifying the variable simultaneously, leading to a race condition.

To prevent this and ensure the counter is incremented consistently, we can use synchronization. Here's how you can synchronize access to the shared variable:

```
static class IncrementTask implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            synchronized (SynchronizationExample.class) {
                sharedCounter++; // Increment the shared counter within
a synchronized block
            }
        }
    }
```

```
}
```

By using `**synchronized**` blocks, we ensure that only one thread can access the sharedCounter at a time, preventing data corruption and ensuring that the counter is incremented correctly.

- To print numbers from 1 to 100 using four different threads where each thread prints a number sequentially

```
public class NumberPrintingExample {
    private static int currentNumber = 1;
    private static final int MAX_NUMBER = 100;
    private static final Object lock = new Object();

    public static void main(String[] args) {
        Thread[] threads = new Thread[4];

        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new NumberPrinter(i + 1));
            threads[i].start();
        }

        try {
            for (Thread thread : threads) {
                thread.join();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    static class NumberPrinter implements Runnable {
        private final int threadNumber;

        public NumberPrinter(int threadNumber) {
            this.threadNumber = threadNumber;
        }

        @Override
        public void run() {
            while (true) {
                synchronized (lock) {
                    if (currentNumber > MAX_NUMBER) {
```

```
                        return; // All numbers printed, exit the thread
                    }

                    if (currentNumber % 4 == threadNumber - 1) {
                        System.out.println("Thread " + threadNumber + ":
" + currentNumber);
                        currentNumber++;
                    }
                }
            }
        }
    }
}
```

This code demonstrates how to use synchronization to coordinate the printing of numbers across multiple threads, ensuring that each thread prints numbers sequentially without overlapping.

Reader-Writer Lock :

The Reader-Writer Problem is a classic synchronization problem in computer science. It involves multiple processes (readers and writers) that access a shared resource, such as a data structure or a database. The problem is to ensure that multiple readers can access the resource simultaneously without any conflicts, but when a writer wants to modify the resource, it should have exclusive access.

Reader-Writer Problem Characteristics:

1. Readers: Multiple readers can access the resource simultaneously.
2. Writers: Writers need exclusive access to the resource. No other readers or writers are allowed during a write operation.
3. Priority: Writers usually have higher priority to prevent readers from starving.

Solving the Reader-Writer Problem:

Solving the reader-writer problem requires synchronization mechanisms to control access to the shared resource. There are several solutions to this problem, but two common ones are the "Reader-Preferred" and "Writer-Preferred" solutions.

Let's explore both solutions with example code:

Reader-Preferred Solution:

In this approach, if a reader is already reading the resource, other readers can join. But if a writer requests access, it has to wait until no readers are reading. This ensures a balance between read and write operations.

Here's Java code illustrating the reader-preferred solution:

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

public class ReaderPreferredExample {
    private static ReentrantReadWriteLock rwLock = new
ReentrantReadWriteLock();
    private static int sharedResource = 0;

    public static void main(String[] args) {
        Thread reader1 = new Thread(new Reader(), "Reader 1");
        Thread writer1 = new Thread(new Writer(), "Writer 1");
        Thread reader2 = new Thread(new Reader(), "Reader 2");

        reader1.start();
        writer1.start();
        reader2.start();
    }

    static class Reader implements Runnable {
        @Override
        public void run() {
            rwLock.readLock().lock();
            System.out.println(Thread.currentThread().getName() + " is
reading: " + sharedResource);
            rwLock.readLock().unlock();
        }
    }

    static class Writer implements Runnable {
        @Override
        public void run() {
            rwLock.writeLock().lock();
            sharedResource++;
            System.out.println(Thread.currentThread().getName() + " is
writing: " + sharedResource);
            rwLock.writeLock().unlock();
        }
    }
}
```

In this example:

- We use a `ReentrantReadWriteLock` for synchronization.
- Readers acquire a read lock while writers acquire a write lock.
- Multiple readers can read concurrently, but writers have exclusive access during a write operation.

Writer-Preferred Solution:

In the writer-preferred approach, writers have a higher priority. When a writer requests access, it blocks all new reader requests until it finishes writing. This can be more efficient in certain scenarios where write operations are less frequent.

Here's a Java example:

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

public class WriterPreferredExample {
    private static Semaphore resourceAccess = new Semaphore(1);
    private static Semaphore readCountAccess = new Semaphore(1);
    private static int readCount = 0;

    public static void main(String[] args) {
        Thread reader1 = new Thread(new Reader(), "Reader 1");
        Thread writer1 = new Thread(new Writer(), "Writer 1");
        Thread reader2 = new Thread(new Reader(), "Reader 2");

        reader1.start();
        writer1.start();
        reader2.start();
    }

    static class Reader implements Runnable {
        @Override
        public void run() {
            try {
                readCountAccess.acquire();
                readCount++;
                if (readCount == 1) {
                    resourceAccess.acquire();
                }
                readCountAccess.release();

                System.out.println(Thread.currentThread().getName() + "
is reading");
```

```
                readCountAccess.acquire();
                readCount--;
                if (readCount == 0) {
                    resourceAccess.release();
                }
                readCountAccess.release();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    static class Writer implements Runnable {
        @Override
        public void run() {
            try {
                resourceAccess.acquire();
                System.out.println(Thread.currentThread().getName() + "
is writing");
                resourceAccess.release();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

In this example:

- We use semaphores for synchronization.
- Writers block new readers during write operations, ensuring they have exclusive access.

Deadlock - is a situation in multithreading where two or more threads are unable to proceed because each is waiting for the other(s) to release a resource or lock. In other words, it's a state where the threads are stuck and can't make any progress, effectively leading to a system-wide standstill. Deadlocks can be challenging to identify and resolve.

A typical deadlock scenario involves the following conditions, known as the four necessary conditions:

1. Mutual Exclusion: Resources that are being accessed are non-shareable. Only one thread can use a resource at a time.

2. Hold and Wait: Threads hold a resource and are waiting for additional resources. If a thread cannot acquire all the resources it needs, it may release the resources it's already holding, but in many cases, it doesn't.

3. No Preemption: Resources cannot be forcibly taken away from a thread; they must be released voluntarily.

4. Circular Wait: A cycle exists in the resource allocation graph, meaning each thread in the cycle is waiting for a resource held by another thread in the cycle.

Here's an example to illustrate deadlock:

```
public class DeadlockExample {
    private static final Object lock1 = new Object();
    private static final Object lock2 = new Object();

    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            synchronized (lock1) {
                System.out.println("Thread 1: Holding lock1...");
                try { Thread.sleep(100); } catch (InterruptedException
e) {}
                System.out.println("Thread 1: Waiting for lock2...");
                synchronized (lock2) {
                    System.out.println("Thread 1: Acquired lock2.");
                }
            }
        });

        Thread thread2 = new Thread(() -> {
            synchronized (lock2) {
                System.out.println("Thread 2: Holding lock2...");
                try { Thread.sleep(100); } catch (InterruptedException
e) {}
                System.out.println("Thread 2: Waiting for lock1...");
                synchronized (lock1) {
                    System.out.println("Thread 2: Acquired lock1.");
                }
            }
        });

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

In this code:

- We have two threads, `thread1` and `thread2`, each acquiring two locks (`lock1` and `lock2`) in a different order.

- Thread 1 locks `lock1` and then attempts to lock `lock2`, while Thread 2 locks `lock2` and then attempts to lock `lock1`.

When you run this code, it's likely to result in a deadlock because each thread holds one lock while waiting for the other lock to be released. The program will appear to hang indefinitely.

To prevent or resolve deadlocks, you can use techniques like:

1. Lock Ordering: Always acquire locks in a consistent order to avoid circular waiting.
2. Timeouts: Implement timeouts to release resources if they're not acquired within a certain time.
3. Resource Allocation Graph: Use tools or algorithms to detect and break deadlock cycles.
4. Avoidance Algorithms: Use algorithms that ensure that conditions for deadlock cannot occur.
5. Recovery: Automatically restart or terminate processes when a deadlock is detected.

Addressing and avoiding deadlocks is an important aspect of concurrent programming to ensure the reliability and stability of multithreaded applications.

Here's an example of how to avoid deadlock by consistently acquiring locks in a specific order:

```
public class DeadlockAvoidanceExample {
    private static final Object lock1 = new Object();
    private static final Object lock2 = new Object();

    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            synchronized (lock1) {
                System.out.println("Thread 1: Holding lock1...");
                try { Thread.sleep(100); } catch (InterruptedException
e) {}
                System.out.println("Thread 1: Trying to acquire
lock2...");
                synchronized (lock2) {
                    System.out.println("Thread 1: Acquired lock2.");
                }
            }
```

```
        });

        Thread thread2 = new Thread(() -> {
            synchronized (lock1) {  // Acquiring lock1 before lock2
                System.out.println("Thread 2: Holding lock1...");
                try { Thread.sleep(100); } catch (InterruptedException
e) {}
                System.out.println("Thread 2: Trying to acquire
lock2...");
                synchronized (lock2) {
                    System.out.println("Thread 2: Acquired lock2.");
                }
            }
        });

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

In this code, both threads consistently acquire `lock1` before attempting to acquire `lock2`. This ensures that the order of lock acquisition is the same for all threads, preventing a circular wait condition and reducing the likelihood of a deadlock.