Deadlocks and Starvation in Java

Explain Deadlocks and Starvation issues in Java.Along with its Solution

Deadlocks in Java

A **deadlock** occurs in Java when two or more threads are waiting for each other's resources indefinitely, preventing further execution. This typically happens when multiple threads acquire locks in different orders.

Example of Deadlock:

```
java


  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 lock 1...");
                  try { Thread.sleep(100); } catch (InterruptedException e) {}
                  System.out.println("Thread 1: Waiting for lock 2...");
                  synchronized (lock2) {
                      System.out.println("Thread 1: Acquired lock 2!");
                  }
              }
          });

          Thread thread2 = new Thread(() -> {
              synchronized (lock2) {
                  System.out.println("Thread 2: Holding lock 2...");
                  try { Thread.sleep(100); } catch (InterruptedException e) {}
                  System.out.println("Thread 2: Waiting for lock 1...");
```

```
                synchronized (lock1) {
                    System.out.println("Thread 2: Acquired lock 1!");
                }
            }
        });

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

In this example, both threads are waiting for each other to release a lock, causing a deadlock.

Solutions to Deadlock:

1. **Lock Ordering**: Always acquire locks in the same order in all threads.
2. **Try-and-Acquire with Timeout**: Use `tryLock()` from the `ReentrantLock` class with a timeout.
3. **Deadlock Detection**: Implement a monitoring mechanism to detect and handle deadlocks.

Starvation in Java

Starvation occurs when a thread is unable to gain regular access to resources and is perpetually denied execution. This usually happens when threads with higher priority monopolize resources or due to unfair scheduling.

Example of Starvation:

```
java


import java.util.concurrent.locks.ReentrantLock;

public class StarvationExample {
    private static final ReentrantLock lock = new ReentrantLock(true); // Fair lock

    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            Thread thread = new Thread(new Worker(), "Thread-" + i);
            thread.start();
        }
    }
```

```
    static class Worker implements Runnable {
        @Override
        public void run() {
            while (true) {
                try {
                    if (lock.tryLock()) {
                        System.out.println(Thread.currentThread().getName() + " acquired the lock.");
                        Thread.sleep(50);
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    lock.unlock();
                }
            }
        }
    }
}
```

In this example, if the lock is not fair (`ReentrantLock(true)`), higher-priority threads might repeatedly acquire the lock, starving lower-priority threads.

Solutions to Starvation:

1. **Fair Locks**: Use `ReentrantLock` with fairness (`new ReentrantLock(true)`).
2. **Thread Priorities**: Avoid excessive use of thread priorities.
3. **Balanced Resource Allocation**: Ensure all threads get fair access to shared resources.

Key Differences:

Aspect	Deadlock	Starvation
Cause	Threads waiting for each other's locks.	Threads unable to access resources.
Behavior	No progress; all threads are stuck.	Affected threads make no progress.
Solution	Lock ordering, timeouts, or detection.	Fair locks, balanced resource allocation.

By addressing these issues with appropriate strategies, you can prevent concurrency problems in Java applications.

Explain Livelocks and Race Conditions issues in Java.Along with its Solution

Livelocks in Java

A **livelock** occurs when two or more threads continuously change their state in response to each other without making progress. Unlike a deadlock, where threads are stuck waiting for resources, in a livelock, threads are active but unable to proceed.

Example of Livelock:

```
java


  public class LivelockExample {
      static class Resource {
          private boolean isLocked = false;

          public synchronized boolean isLocked() {
              return isLocked;
          }

          public synchronized void setLocked(boolean isLocked) {
              this.isLocked = isLocked;
          }
      }

      public static void main(String[] args) {
          Resource resource1 = new Resource();
          Resource resource2 = new Resource();

          Thread thread1 = new Thread(() -> {
              while (true) {
                  if (!resource1.isLocked()) {
                      resource1.setLocked(true);
                      System.out.println("Thread 1 locked resource 1");

                      if (resource2.isLocked()) {
                          System.out.println("Thread 1 released resource 1");
                          resource1.setLocked(false);
                          continue;
                      }

                      System.out.println("Thread 1 locked resource 2");
```

```
                    break;
                }
            }
        });

        Thread thread2 = new Thread(() -> {
            while (true) {
                if (!resource2.isLocked()) {
                    resource2.setLocked(true);
                    System.out.println("Thread 2 locked resource 2");

                    if (resource1.isLocked()) {
                        System.out.println("Thread 2 released resource 2");
                        resource2.setLocked(false);
                        continue;
                    }

                    System.out.println("Thread 2 locked resource 1");
                    break;
                }
            }
        });

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

In this example, both threads release and acquire resources in response to each other, leading to no progress.

Solutions to Livelock:

1. **Timeouts**: Use timeouts or backoff strategies to avoid continuous retries.
2. **Order of Operations**: Define a strict order in which resources are acquired or tasks are executed.
3. **Randomization**: Add randomness to decision-making to reduce contention.

Race Conditions in Java

A **race condition** occurs when multiple threads access shared resources simultaneously, and the outcome depends on the timing of their execution. This can lead to inconsistent or incorrect results.

Example of Race Condition:

```java
public class RaceConditionExample {
    private int counter = 0;

    public void increment() {
        counter++;
    }

    public int getCounter() {
        return counter;
    }

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

        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                example.increment();
            }
        });

        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                example.increment();
            }
        });

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

        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

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

In this example, the `**increment()**` method is not thread-safe, so the final counter may not be `**2000**` due to overlapping updates from multiple threads.

Solutions to Race Condition:

1. **Synchronization**: Use the `synchronized` keyword to ensure only one thread accesses the critical section at a time.

```
java


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

2. **Atomic Variables**: Use classes from the `java.util.concurrent.atomic` package, such as `AtomicInteger`.

```
java


  private AtomicInteger counter = new AtomicInteger();

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

3. **Locks**: Use explicit locks like `ReentrantLock` to control access to shared resources.

```
java


  private ReentrantLock lock = new ReentrantLock();

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

Key Differences:

Aspect	Livelock	Race Condition
Cause	Threads respond to each other without making progress.	Shared resource is accessed without proper synchronization.
Behavior	Threads remain active but unable to proceed.	Unpredictable and incorrect results due to timing.
Solution	Timeouts, backoff, ordering, or randomization.	Synchronization, atomic variables, or explicit locks.

By addressing these issues, you can ensure better concurrency and reliability in Java programs.