## **Deadlock Scenario**

**What is a Deadlock?**

A **deadlock** occurs when two or more threads block each other forever because each thread is waiting for a resource that the other thread is holding.

**Why Does Deadlock Happen?**

Deadlock happens when:

* Threads need multiple resources to complete their tasks.
* Threads lock one resource and wait for another, but the second resource is already locked by another thread.
* None of the threads can continue, causing the program to freeze.

### **Real-World Analogy**

Imagine two people trying to pass through a narrow corridor from opposite sides:

* Person A enters from one end, and Person B enters from the other end.
* They both stop in the middle, blocking each other.
* Neither can move forward because both are waiting for the other to step aside.

This situation is a deadlock.

**Code Example of Deadlock**

****// Writing code for showing deadlock condition in java program using synchronized block and object locking.

// Writing the code without creating a thread using the thread class.

public class DeadLock extends Thread {

// Define two resources as static objects

private static final Object resource1 = new Object();

private static final Object resource2 = new Object();

public void run() {

// Thread 1 locks resource1 first

synchronized (resource1) {

System.out.println("Thread 1: Locked Resource 1");

try {

// Simulate some work with Resource 1

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

// Thread 1 tries to lock resource2

synchronized (resource2) {

System.out.println("Thread 1: Locked Resource 2");

}

}

// Thread 2 locks resource2 first

synchronized (resource2) {

System.out.println("Thread 2: Locked Resource 2");

try {

// Simulate some work with Resource 2

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

// Thread 2 tries to lock resource1

synchronized (resource1) {

System.out.println("Thread 2: Locked Resource 1");

}

}

}

public static void main(String[] args) {

// Start two threads

DeadLock d1 = new DeadLock();

d1.start();

DeadLock d2 = new DeadLock();

d2.start();

}

}

**Above Code Explanation:**

The code is in a deadlock condition because two threads are trying to acquire locks on two resources (resource1 and resource2) in opposite order. Here is the sequence of events leading to the deadlock:

* **Thread 1** locks resource1 and then tries to lock resource2.
* **Thread 2** locks resource2 and then tries to lock resource1.

Since both threads are holding one resource and waiting for the other, they end up in a deadlock, where neither thread can proceed.

### 

### **How to Avoid Deadlocks?**

**Lock Resources in a Fixed Order:** Always lock resources in the same sequence.

For example, if threads must lock **resource1** and then **resource2**, ensure all threads follow this order.

**Example:**

synchronized (resource1) {

synchronized (resource2) {

// Do work

}

}



## **Daemon Threads**

### **Let’s begin with, What is a Daemon Thread?**

A **daemon thread** is a background thread that runs in the JVM to perform low-priority tasks. Unlike regular threads, daemon threads don’t block the program from shutting down. When all non-daemon threads finish, the JVM automatically terminates the daemon threads.

### **Let’s see, Why Were Daemon Threads Introduced?**

Daemon threads are used for tasks that:

* Run continuously in the background.
* Don’t need to finish if the program exits.

### **Let’s see a Real-World Analogy as a JANITOR**

Daemon threads are like a janitor in a shopping mall:

* The janitor cleans the mall in the background while customers are shopping.
* If all customers leave and the mall closes, the janitor stops working even if cleaning isn’t finished.

### **Examples of Daemon Threads in Java**

* **Garbage Collector**: Cleans up unused objects in memory.
* **Timer Threads**: Schedule tasks to run periodically.

### **Let’s see, How to Create a Daemon Thread**

To make a thread a daemon, call setDaemon(true) before starting the thread.

**Code Example:**

public class DaemonThreadExample {

public static void main(String[] args) {

// Create a daemon thread by passing a Runnable to a Thread

DaemonTask daemonTask = new DaemonTask();

Thread daemonThread = new Thread(daemonTask);

// Set the thread as a daemon

daemonThread.setDaemon(true);

// Start the daemon thread

daemonThread.start();

// Main thread simulates some work and then ends

try {

System.out.println("Main thread is running...");

Thread.sleep(2000); // Main thread sleeps for 2 seconds

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Main thread is finished. Daemon thread will stop.");

}

}

// A simple class implementing Runnable to define the daemon thread's task

class DaemonTask implements Runnable {

@Override

public void run() {

while (true) {

System.out.println("Daemon thread is running in the background...");

try {

Thread.sleep(500); // Sleep for 500ms

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**Output:**

* The daemon thread keeps running in the background until the main thread finishes.
* Once the main thread exits, the daemon thread also stops automatically.

Main thread is running...

Daemon thread is running in the background...

Daemon thread is running in the background...

Daemon thread is running in the background...

Daemon thread is running in the background...

Main thread is finished. Daemon thread will stop.

### **Practical Usage of Daemon Threads**

#### **Auto-Saving in Applications**

Imagine a text editor that saves your work automatically every minute. A daemon thread can handle this without interrupting the user.

**Code Example:**

public class AutoSaveExample {

public static void main(String[] args) {

// Create a thread for auto-save functionality

AutoSaveTask autoSaveTask = new AutoSaveTask();

Thread autoSaveThread = new Thread(autoSaveTask);

// Set the thread as a daemon

autoSaveThread.setDaemon(true);

// Start the auto-save thread

autoSaveThread.start();

// Simulate user activity in the main thread

try {

System.out.println("User is working...");

Thread.sleep(5000); // Simulate 5 seconds of user work

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("User closed the application. Auto-save will stop.");

}

}

// Task for auto-saving a document

class AutoSaveTask implements Runnable {

@Override

public void run() {

while (true) {

System.out.println("Auto-saving document...");

try {

Thread.sleep(1000); // Save every 1 second

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

### 

**Output:**

User is working...

Auto-saving document...

Auto-saving document...

Auto-saving document...

Auto-saving document...

Auto-saving document...

User closed the application. Auto-save will stop.

### 

### Finally let’s see the, **some Key Points About Daemon Threads**

* **Daemon threads run in the background** and stop automatically when all user threads (non-daemon threads) finish.
* **Use for low-priority tasks** like logging, memory cleanup, or periodic checks.
* Always set a thread as a daemon using setDaemon(true) **before starting the thread**. If done after starting, it will throw IllegalThreadStateException.