

**PROJECT REPORT**

**JAVA PROGRAMMING FUNDAMENTALS**

**(EBDS22ET2)**

2024-2025(EVEN SEMESTER)

**DEPARTMENT OF B.Tech DS & AI (E&T)**

**COURSE :** B-TECH CSE-DS (AI)

**YEAR/SEM/SEC :** 1styear / 2nd Sem / AC1

**PROJECT TITLE : LIFE CYCLE OF THREADS IN JAVA**



**BONAFIDE CERTIFICATE**

**JAVA PROGRAMMING FUNDAMENTALS**

**DEPARTMENT OF B.Tech DS & AI (E&T)**

Certified that this project report **“Life Cycle Of Threads In Java”** is confirmed work of **Arcot Nandha Kumar, Manishwaran. R, M. Naveen Kumar** I-year B-Tech CSE- DS(AI) in **JAVA PROGRAMMING FUNDAMENTALS (EBDS22ET2)** who carried out the project work under the supervision

Signature of Lab-in-Charge Signature of Head of Dept

Submitted for the Practical Examination held on

Internal Examiner External Examiner

# ABSTRACT

.

In modern software development, especially in the realm of concurrent programming, thread management is a cornerstone for optimizing performance and resource utilization. Java, as a widely adopted object-oriented programming language, provides robust support for multithreading, allowing developers to build applications capable of performing multiple tasks concurrently. A fundamental concept in this paradigm is the *thread life cycle*, which outlines the various stages a thread transitions through during its existence—from creation to termination. Understanding this life cycle is essential not only for efficient thread handling but also for diagnosing thread-related issues such as deadlocks, race conditions, and unexpected behaviors during runtime.

This study explores the different states that a thread undergoes in Java, specifically: **New**, **Runnable**, **Running**, **Blocked/Waiting**, and **Terminated**. Each of these states plays a distinct role in managing execution flow and system resources. A thread begins in the *New* state when it is instantiated but not yet started. Upon invoking the start() method, the thread enters the *Runnable* state, indicating its readiness for execution. However, actual execution begins only when the thread scheduler allocates CPU time, transitioning it into the *Running* state. During execution, a thread may enter *Blocked*, *Waiting*, or *Timed Waiting* states based on operations like resource locking or deliberate delays (e.g., sleep() or join() methods). Eventually, after completing its task or being forcibly stopped, the thread reaches the *Terminated* state.

To elucidate these transitions, the paper presents a Java-based demonstration program titled **ThreadLifeCycleDemo**, wherein a custom thread class MyThread extends the Thread class. The program vividly tracks the thread’s state at various execution points using getState() method calls. It highlights state changes such as NEW upon thread instantiation, RUNNABLE after invoking start(), TIMED\_WAITING during Thread.sleep(), and TERMINATED post completion of execution. The practical illustration provides developers with a clear understanding of thread dynamics and emphasizes the importance of lifecycle awareness in writing efficient, responsive, and thread-safe code.

The expected output of the demonstration aligns closely with real-world thread behavior, thus reinforcing the theoretical concepts with observable execution patterns. By mapping thread states to program outputs, this exploration offers a hands-on approach to mastering thread operations in Java. The understanding gained through this exercise serves as a foundational skill for tackling more advanced concurrency mechanisms such as synchronization, thread pooling, and executor frameworks. In conclusion, the thread life cycle is not merely a theoretical model but a practical guide for writing robust multithreaded Java applications that can gracefully manage resources and handle real-time processing demands.

**INTRODUCTION TO JAVA:**

Java is a high-level, object-oriented programming language that has become a cornerstone of modern software development due to its platform independence, strong memory management, and extensive standard library. Designed with the principle of "write once, run anywhere," Java has been widely adopted across industries for developing everything from desktop applications and web services to large-scale enterprise systems and embedded solutions. One of Java’s most powerful features is its built-in support for **multithreading**, which allows multiple threads of execution to run concurrently within a single program. This makes Java particularly suitable for creating responsive, high-performance, and scalable applications in today’s multi-core computing environments.

At the heart of Java's multithreading capabilities lies the **thread life cycle**, a fundamental concept that governs how threads are created, managed, executed, and terminated. Threads enable the simultaneous execution of code segments, which is essential for tasks such as background processing, real-time computation, and asynchronous data handling. Understanding the thread life cycle is critical for developers seeking to harness Java's full potential in building efficient and concurrent applications. A thread in Java transitions through various states—**New**, **Runnable**, **Running**, **Blocked/Waiting**, and **Terminated**—each representing a unique phase in its execution journey.

To better understand these transitions, a Java-based demonstration program titled *ThreadLifeCycleDemo* illustrates the practical behavior of threads using the Thread class and its associated methods. Through the implementation of a custom thread class MyThread, the program showcases real-time state changes such as NEW, RUNNABLE, TIMED\_WAITING, and TERMINATED, using calls to the getState() method. This hands-on example offers valuable insight into how thread scheduling, sleep delays, and task completion influence thread behavior in a controlled environment.

By mastering the life cycle of threads, developers not only improve the performance of their Java applications but also gain the ability to avoid common concurrency pitfalls like race conditions, deadlocks, and resource contention. This foundational knowledge paves the way for exploring advanced concepts such as synchronized blocks, thread pooling, the java.util.concurrent package, and executor services. In essence, Java’s support for multithreading and its well-defined thread life cycle model provide a powerful framework for building dynamic, responsive, and reliable software systems.

# PROJECT TITLE

**AIM:**

The aim of this project is to demonstrate and analyze the various states in the life cycle of a thread in Java by implementing a practical Java program that visually tracks state transitions using real-time thread operations.

**ALGORITHMS:**

**Step 1**: Define a class that extends the Thread class.

**Step 2**: Override the run() method to include thread execution logic.

**Step 3**: Inside the run() method, print a message indicating the thread is running.

**Step 4**: Use Thread.sleep() to simulate a timed waiting state.

**Step 5**: After the sleep period, print a message indicating the thread is completing its task.

**Step 6**: In the main() method, create an instance of the custom thread class.

**Step 7**: Print the initial state of the thread using getState().

**Step 8**: Start the thread using the start() method.

**Step 9**: Immediately print the thread’s state after calling start().

**Step 10**: Pause the main thread using Thread.sleep() to allow observation of the thread's state.

**Step 11**: Print the state of the custom thread while it is sleeping.

**Step 12**: Use join() to wait for the custom thread to finish execution.

**Step 13**: After the thread completes, print its final state.

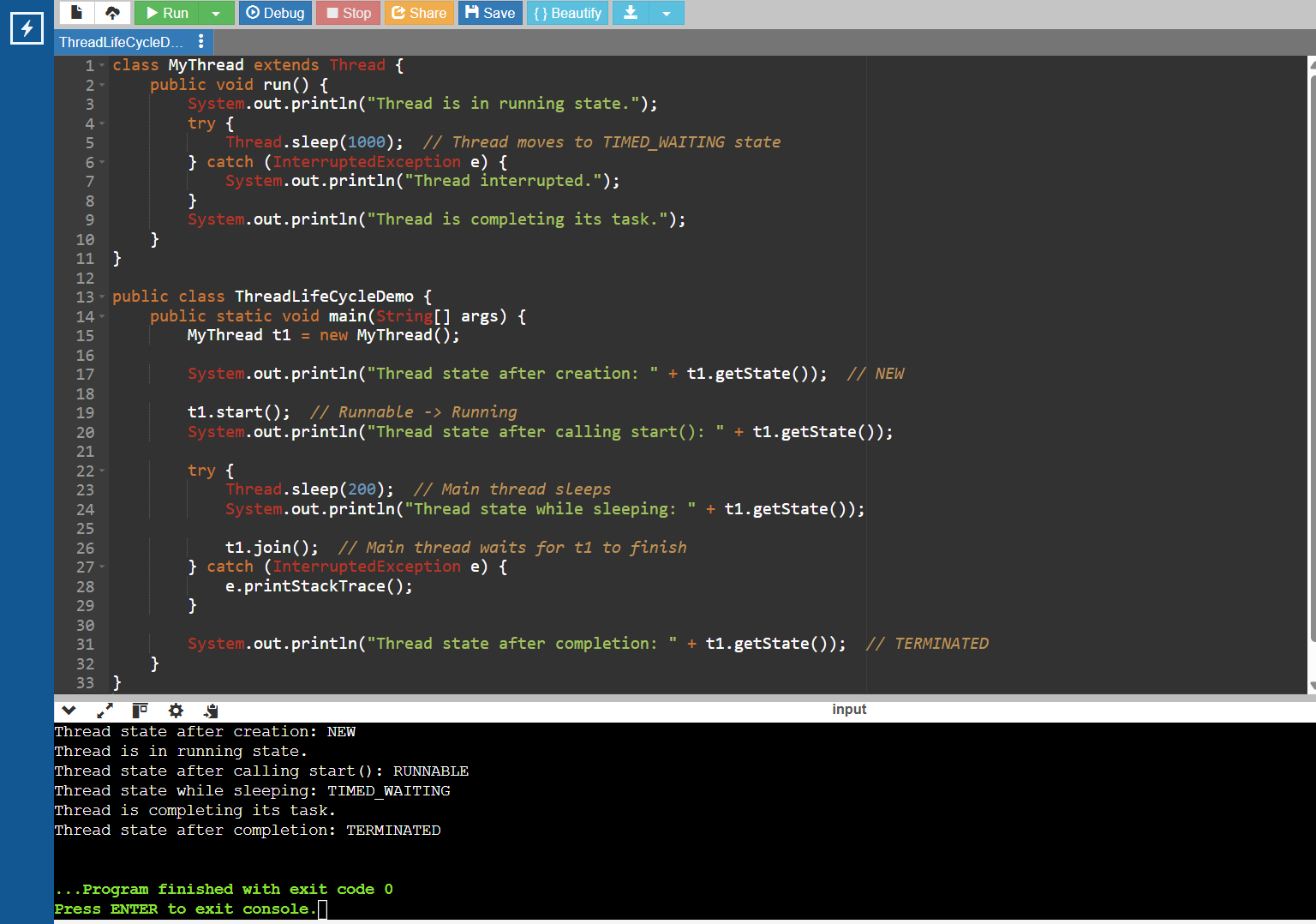
**SOURCE CODE:**

class MyThread extends Thread {  
 public void run() {  
 System.out.println("Thread is in running state.");  
 try {  
 Thread.sleep(1000); // Thread moves to TIMED\_WAITING state  
 } catch (InterruptedException e) {  
 System.out.println("Thread interrupted.");  
 }  
 System.out.println("Thread is completing its task.");  
 }  
}  
  
public class ThreadLifeCycleDemo {  
 public static void main(String[] args) {  
 MyThread t1 = new MyThread();  
  
 System.out.println("Thread state after creation: " + t1.getState()); // NEW  
  
 t1.start(); // Runnable -> Running  
 System.out.println("Thread state after calling start(): " + t1.getState());

try {  
 Thread.sleep(200); // Main thread sleeps  
 System.out.println("Thread state while sleeping: " + t1.getState());  
  
 t1.join(); // Main thread waits for t1 to finish  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 System.out.println("Thread state after completion: " + t1.getState()); // TERMINATED  
 }  
}

**OUTPUT:**

Thread state after creation: NEW  
Thread state after calling start(): RUNNABLE  
Thread is in running state.  
Thread state while sleeping: TIMED\_WAITING  
Thread is completing its task.  
Thread state after completion: TERMINATED

****

**RESULT:**

The program successfully demonstrates the life cycle of a thread in Java. It prints the thread’s state at different execution points, confirming transitions through NEW, RUNNABLE, TIMED\_WAITING, and TERMINATED states, as expected.