Introduction to Java and Core OOP Concepts

Dr. Ratnesh Prasad Srivastava
Department of CSIT, GGV, Bilaspur (C.G.)

Academic Year: 2026-27

Course Information

Course Code	CIUDMJT1
Course Title	Object-Oriented Programming with Java
Credit Hours	3-0-3 (3 Lecture, 0 Tutorial, 3 Practical)
Prerequisites	Programming Fundamentals
Textbook	"Java: The Complete Reference" by Herbert Schildt
Reference	"Head First Java" by Kathy Sierra and Bert Bates

Contents

1 Unit IV: Multithreading and Advanced Java Concepts

1.1 Learning Objectives

- Understand the fundamentals of multithreading and concurrency
- Master thread creation using Thread class and Runnable interface
- Comprehend thread lifecycle and state transitions
- Implement thread synchronization using synchronized keyword
- Understand inter-thread communication using wait/notify
- Apply multithreading concepts to solve real-world problems
- Analyze thread safety issues and implement solutions

2 Introduction to Multithreading

2.1 What is a Thread?

Thread Definition

A **thread** is a lightweight sub-process, the smallest unit of processing. Threads are independent, concurrent execution paths within a program that share the same memory space. Multiple threads can exist within the same process and share resources like memory, while processes are isolated from each other.

2.2 Single-threaded vs Multi-threaded Programs

```
1  public class ThreadIntroduction {
2
3      // ==================== SINGLE-THREADED EXECUTION
           ====================
4      public static void singleThreadedExample() {
5          System.out.println("=== SINGLE-THREADED EXECUTION ===");
6          System.out.println("Main thread started: " + Thread.
               currentThread().getName());
7
8          long startTime = System.currentTimeMillis();
9
10         // Sequential execution - one task after another
11         task1();
12         task2();
13         task3();
14
15         long endTime = System.currentTimeMillis();
16         System.out.println("Total time taken: " + (endTime - startTime)
               + " ms");
17         System.out.println("Main thread completed\n");
18     }
```

```
19
20      private static void task1() {
21          System.out.println("Task 1 started by: " + Thread.currentThread
                ().getName());
22          try {
23              Thread.sleep(1000); // Simulate work
24          } catch (InterruptedException e) {
25              e.printStackTrace();
26          }
27          System.out.println("Task 1 completed");
28      }
29
30      private static void task2() {
31          System.out.println("Task 2 started by: " + Thread.currentThread
                ().getName());
32          try {
33              Thread.sleep(1500); // Simulate work
34          } catch (InterruptedException e) {
35              e.printStackTrace();
36          }
37          System.out.println("Task 2 completed");
38      }
39
40      private static void task3() {
41          System.out.println("Task 3 started by: " + Thread.currentThread
                ().getName());
42          try {
43              Thread.sleep(800); // Simulate work
44          } catch (InterruptedException e) {
45              e.printStackTrace();
46          }
47          System.out.println("Task 3 completed");
48      }
49
50      // ==================== MULTI-THREADED EXECUTION
            ====================
51      public static void multiThreadedExample() {
52          System.out.println("=== MULTI-THREADED EXECUTION ===");
53          System.out.println("Main thread started: " + Thread.
                currentThread().getName());
54
55          long startTime = System.currentTimeMillis();
56
57          // Create and start multiple threads
58          Thread thread1 = new Thread(() -> task1(), "Worker-1");
59          Thread thread2 = new Thread(() -> task2(), "Worker-2");
60          Thread thread3 = new Thread(() -> task3(), "Worker-3");
61
62          thread1.start();
63          thread2.start();
64          thread3.start();
65
66          // Wait for all threads to complete
67          try {
68              thread1.join();
69              thread2.join();
70              thread3.join();
71          } catch (InterruptedException e) {
```

```
72              e.printStackTrace();
73          }
74
75          long endTime = System.currentTimeMillis();
76          System.out.println("Total time taken: " + (endTime - startTime)
                + " ms");
77          System.out.println("Main thread completed\n");
78      }
79
80      // ==================== REAL-WORLD ANALOGY ====================
81      public static void restaurantAnalogy() {
82          System.out.println("=== RESTAURANT ANALOGY ===");
83          System.out.println("Single-threaded Restaurant:");
84          System.out.println("  - One waiter does everything: takes order
                , cooks, serves");
85          System.out.println("  - Customers wait a long time");
86          System.out.println("  - Inefficient use of resources");
87
88          System.out.println("\nMulti-threaded Restaurant:");
89          System.out.println("  - Host: Greets customers (main thread)");
90          System.out.println("  - Waiter Thread: Takes orders");
91          System.out.println("  - Chef Thread: Prepares food");
92          System.out.println("  - Busser Thread: Cleans tables");
93          System.out.println("  - Faster service, better resource
                utilization");
94      }
95
96      // ==================== ADVANTAGES OF MULTITHREADING
            ====================
97      public static void demonstrateAdvantages() {
98          System.out.println("\n=== ADVANTAGES OF MULTITHREADING ===");
99
100         // 1. Responsiveness
101         System.out.println("\n1. Responsiveness:");
102         System.out.println("   - GUI applications remain responsive");
103         System.out.println("   - One thread handles UI, another does
                background work");
104         System.out.println("   - Example: Download file while editing
                document");
105
106         // 2. Resource Sharing
107         System.out.println("\n2. Resource Sharing:");
108         System.out.println("   - Threads share memory space");
109         System.out.println("   - No need for inter-process
                communication (IPC)");
110         System.out.println("   - Efficient communication between
                threads");
111
112         // 3. Economy
113         System.out.println("\n3. Economy:");
114         System.out.println("   - Creating threads is cheaper than
                processes");
115         System.out.println("   - Context switching between threads is
                faster");
116         System.out.println("   - Less memory overhead");
117
118         // 4. Utilization of Multiprocessor Architecture
```

```
119         System.out.println("\n4. Utilization of Multiprocessor
                Architecture:");
120         System.out.println("   - Multiple threads can run on multiple
                CPUs");
121         System.out.println("   - True parallel execution");
122         System.out.println("   - Better CPU utilization");
123
124         // 5. Simplified Modeling
125         System.out.println("\n5. Simplified Modeling:");
126         System.out.println("   - Natural modeling of concurrent
                activities");
127         System.out.println("   - Example: Web server handling multiple
                clients");
128         System.out.println("   - Each client connection = separate
                thread");
129     }
130
131     // ==================== PROCESS VS THREAD ====================
132     public static void processVsThread() {
133         System.out.println("\n=== PROCESS VS THREAD ===");
134
135         System.out.println("\nProcess:");
136         System.out.println("  - Independent execution unit");
137         System.out.println("  - Has its own memory space");
138         System.out.println("  - Heavyweight (more resources)");
139         System.out.println("  - Inter-process communication (IPC)
                required");
140         System.out.println("  - Created using OS system calls");
141
142         System.out.println("\nThread:");
143         System.out.println("  - Lightweight sub-process");
144         System.out.println("  - Shares memory with other threads in
                same process");
145         System.out.println("  - Lightweight (less resources)");
146         System.out.println("  - Direct communication through shared
                memory");
147         System.out.println("  - Created within program using Thread
                class");
148
149         System.out.println("\nMemory Layout Comparison:");
150         System.out.println("Process: [Code][Data][Heap][Stack]");
151         System.out.println("Threads in Process: [Code][Data][Heap][
                Stack1][Stack2][Stack3]");
152     }
153
154     // ==================== MAIN METHOD ====================
155     public static void main(String[] args) {
156         System.out.println("=== INTRODUCTION TO MULTITHREADING ===\n");
157
158         // Process vs Thread
159         processVsThread();
160
161         // Real-world analogy
162         restaurantAnalogy();
163
164         // Advantages of multithreading
165         demonstrateAdvantages();
166
```

```
        // Single-threaded execution
        singleThreadedExample();

        // Multi-threaded execution
        multiThreadedExample();

        System.out.println("=== THREAD SCHEDULING ===");
        System.out.println("\nThread Scheduler:");
        System.out.println("  - Part of JVM/OS that decides which thread runs");
        System.out.println("  - Uses scheduling algorithms (Round Robin, Priority)");
        System.out.println("  - Can be preemptive or cooperative");

        System.out.println("\n=== DAEMON THREADS ===");
        System.out.println("Daemon Threads:");
        System.out.println("  - Background threads (GC, auto-save)");
        System.out.println("  - Don't prevent JVM from exiting");
        System.out.println("  - Created using setDaemon(true)");

        System.out.println("\n=== THREAD PRIORITIES ===");
        System.out.println("Thread Priorities (1-10):");
        System.out.println("  - MIN_PRIORITY: 1");
        System.out.println("  - NORM_PRIORITY: 5 (default)");
        System.out.println("  - MAX_PRIORITY: 10");
        System.out.println("  - Higher priority threads get preference");

        // Quick demonstration
        System.out.println("\n=== QUICK DEMONSTRATION ===");
        Thread mainThread = Thread.currentThread();
        System.out.println("Current Thread: " + mainThread.getName());
        System.out.println("Thread ID: " + mainThread.getId());
        System.out.println("Priority: " + mainThread.getPriority());
        System.out.println("Is Daemon: " + mainThread.isDaemon());
        System.out.println("Thread Group: " + mainThread.getThreadGroup().getName());
    }
}
```

Listing 1: Single-threaded vs Multi-threaded Execution

ThreadIntroduction Program Output

```
=== INTRODUCTION TO MULTITHREADING ===

=== PROCESS VS THREAD ===

Process:
  - Independent execution unit
  - Has its own memory space
  - Heavyweight (more resources)
  - Inter-process communication (IPC) required
  - Created using OS system calls
```

```
Thread:
  - Lightweight sub-process
  - Shares memory with other threads in same process
  - Lightweight (less resources)
  - Direct communication through shared memory
  - Created within program using Thread class

Memory Layout Comparison:
Process: [Code][Data][Heap][Stack]
Threads in Process: [Code][Data][Heap][Stack1][Stack2][Stack3]

=== RESTAURANT ANALOGY ===
Single-threaded Restaurant:
  - One waiter does everything: takes order, cooks, serves
  - Customers wait a long time
  - Inefficient use of resources

Multi-threaded Restaurant:
  - Host: Greets customers (main thread)
  - Waiter Thread: Takes orders
  - Chef Thread: Prepares food
  - Busser Thread: Cleans tables
  - Faster service, better resource utilization

=== ADVANTAGES OF MULTITHREADING ===

1. Responsiveness:
   - GUI applications remain responsive
   - One thread handles UI, another does background work
   - Example: Download file while editing document

2. Resource Sharing:
   - Threads share memory space
   - No need for inter-process communication (IPC)
   - Efficient communication between threads

3. Economy:
   - Creating threads is cheaper than processes
   - Context switching between threads is faster
   - Less memory overhead

4. Utilization of Multiprocessor Architecture:
   - Multiple threads can run on multiple CPUs
   - True parallel execution
   - Better CPU utilization

5. Simplified Modeling:
```

```
   - Natural modeling of concurrent activities
   - Example: Web server handling multiple clients
   - Each client connection = separate thread

=== SINGLE-THREADED EXECUTION ===
Main thread started: main
Task 1 started by: main
Task 1 completed
Task 2 started by: main
Task 2 completed
Task 3 started by: main
Task 3 completed
Total time taken: 3308 ms
Main thread completed

=== MULTI-THREADED EXECUTION ===
Main thread started: main
Task 1 started by: Worker-1
Task 2 started by: Worker-2
Task 3 started by: Worker-3
Task 3 completed
Task 1 completed
Task 2 completed
Total time taken: 1504 ms
Main thread completed

=== THREAD SCHEDULING ===

Thread Scheduler:
  - Part of JVM/OS that decides which thread runs
  - Uses scheduling algorithms (Round Robin, Priority)
  - Can be preemptive or cooperative

=== DAEMON THREADS ===
Daemon Threads:
  - Background threads (GC, auto-save)
  - Don't prevent JVM from exiting
  - Created using setDaemon(true)

=== THREAD PRIORITIES ===
Thread Priorities (1-10):
  - MIN_PRIORITY: 1
  - NORM_PRIORITY: 5 (default)
  - MAX_PRIORITY: 10
  - Higher priority threads get preference

=== QUICK DEMONSTRATION ===
```

```
Current Thread: main
Thread ID: 1
Priority: 5
Is Daemon: false
Thread Group: main
```

3 Creating Threads

3.1 Extending Thread Class

```java
public class ExtendingThreadClass {

    // ==================== BASIC THREAD EXTENSION ====================
    static class SimpleThread extends Thread {
        private final String threadName;

        public SimpleThread(String name) {
            this.threadName = name;
            System.out.println("Creating thread: " + threadName);
        }

        @Override
        public void run() {
            System.out.println("Thread " + threadName + " is running");

            try {
                // Simulate some work
                for (int i = 1; i <= 5; i++) {
                    System.out.println(threadName + ": Count " + i);
                    Thread.sleep(500); // Pause for 500ms
                }
            } catch (InterruptedException e) {
                System.out.println(threadName + " interrupted");
            }

            System.out.println("Thread " + threadName + " exiting");
        }
    }

    // ==================== THREAD WITH CUSTOM FUNCTIONALITY
       ====================
    static class CounterThread extends Thread {
        private final String threadName;
        private final int countLimit;

        public CounterThread(String name, int limit) {
            this.threadName = name;
            this.countLimit = limit;
            setName(name); // Set thread name
        }

        @Override
        public void run() {
```

```
43              System.out.println(threadName + " started with priority: "
                    + getPriority());
44
45              int sum = 0;
46              for (int i = 1; i <= countLimit; i++) {
47                  sum += i;
48
49                  if (i % 1000000 == 0) {
50                      System.out.println(threadName + ": Processed " + i
                            + " numbers");
51
52                      // Demonstrate thread control methods
53                      if (i == 3000000) {
54                          System.out.println(threadName + ": Yielding to
                                other threads");
55                          Thread.yield(); // Hint to scheduler
56                      }
57
58                      // Check if thread is interrupted
59                      if (Thread.interrupted()) {
60                          System.out.println(threadName + ": Interruption
                                detected, cleaning up...");
61                          break;
62                      }
63                  }
64              }
65
66              System.out.println(threadName + ": Sum of first " +
                    countLimit +
67                                 " numbers = " + sum);
68              System.out.println(threadName + " completed");
69          }
70      }
71
72      // ==================== THREAD WITH EXCEPTION HANDLING
            ====================
73      static class ExceptionHandlingThread extends Thread {
74          private final String threadName;
75
76          public ExceptionHandlingThread(String name) {
77              this.threadName = name;
78          }
79
80          @Override
81          public void run() {
82              System.out.println(threadName + " started");
83
84              try {
85                  // Simulate work that might throw exceptions
86                  for (int i = 0; i < 10; i++) {
87                      System.out.println(threadName + ": Working on step
                            " + (i + 1));
88                      Thread.sleep(200);
89
90                      // Simulate an error condition
91                      if (i == 5) {
92                          throw new RuntimeException("Simulated error in
                                " + threadName);
```

```
                    }
                }
            } catch (InterruptedException e) {
                System.out.println(threadName + " was interrupted: " +
                    e.getMessage());
            } catch (RuntimeException e) {
                System.out.println(threadName + " caught exception: " +
                     e.getMessage());
                // Handle exception but don't rethrow - thread will
                    complete
            } finally {
                System.out.println(threadName + " in finally block -
                    cleaning up");
            }

            System.out.println(threadName + " completed execution");
        }
    }

    // ==================== THREAD WITH PRIORITIES ====================
    static class PriorityDemoThread extends Thread {
        private final String threadName;
        private int count = 0;

        public PriorityDemoThread(String name) {
            this.threadName = name;
        }

        @Override
        public void run() {
            System.out.println(threadName + " started with priority " +
                 getPriority());

            // Count as fast as possible for 2 seconds
            long startTime = System.currentTimeMillis();
            while (System.currentTimeMillis() - startTime < 2000) {
                count++;
                // Add small delay to prevent overwhelming CPU
                for (int i = 0; i < 1000; i++) {
                    // Busy wait
                }
            }

            System.out.println(threadName + " completed. Count: " +
                count +
                               ", Priority: " + getPriority());
        }

        public int getCount() {
            return count;
        }
    }

    // ==================== DAEMON THREAD DEMONSTRATION
        ====================
    static class DaemonThread extends Thread {
        public DaemonThread(String name) {
            super(name);
```

```
144             setDaemon(true); // Mark as daemon thread
145         }
146 
147         @Override
148         public void run() {
149             System.out.println(getName() + " started. Is Daemon: " +
                    isDaemon());
150 
151             try {
152                 // Daemon thread runs in background
153                 for (int i = 1; i <= 10; i++) {
154                     System.out.println(getName() + ": Background task "
                            + i);
155                     Thread.sleep(1000);
156                 }
157             } catch (InterruptedException e) {
158                 System.out.println(getName() + " interrupted");
159             }
160 
161             System.out.println(getName() + " completed (may not reach
                    this if main thread ends)");
162         }
163     }
164 
165     // ==================== PRACTICAL EXAMPLE: DOWNLOAD MANAGER
            ====================
166     static class DownloadThread extends Thread {
167         private final String fileName;
168         private final int fileSizeMB;
169         private int downloadedMB = 0;
170         private volatile boolean isPaused = false;
171         private volatile boolean isCancelled = false;
172 
173         public DownloadThread(String fileName, int fileSizeMB) {
174             super("Download-" + fileName);
175             this.fileName = fileName;
176             this.fileSizeMB = fileSizeMB;
177         }
178 
179         @Override
180         public void run() {
181             System.out.println(getName() + ": Starting download of " +
                    fileName +
182                                " (" + fileSizeMB + " MB)");
183 
184             try {
185                 for (int i = 1; i <= fileSizeMB && !isCancelled; i++) {
186                     // Check if paused
187                     while (isPaused && !isCancelled) {
188                         Thread.sleep(100);
189                     }
190 
191                     if (isCancelled) {
192                         System.out.println(getName() + ": Download
                                cancelled");
193                         return;
194                     }
195 
```

```
                    // Simulate downloading 1MB
                    Thread.sleep(100); // 100ms per MB
                    downloadedMB = i;

                    if (i % 10 == 0 || i == fileSizeMB) {
                        int percent = (i * 100) / fileSizeMB;
                        System.out.println(getName() + ": " + percent +
                            "% completed (" +
                                    i + "/" + fileSizeMB + " MB)")
                                        ;
                    }
                }

                if (!isCancelled) {
                    System.out.println(getName() + ": Download
                       completed successfully!");
                }

            } catch (InterruptedException e) {
                System.out.println(getName() + ": Download interrupted"
                   );
            }
        }

        public void pauseDownload() {
            isPaused = true;
            System.out.println(getName() + ": Download paused");
        }

        public void resumeDownload() {
            isPaused = false;
            System.out.println(getName() + ": Download resumed");
        }

        public void cancelDownload() {
            isCancelled = true;
            System.out.println(getName() + ": Download cancellation
               requested");
        }

        public int getProgress() {
            return (downloadedMB * 100) / fileSizeMB;
        }
    }

    // ==================== THREAD METHODS DEMONSTRATION
       ====================
    public static void demonstrateThreadMethods() {
        System.out.println("\n=== THREAD METHODS DEMONSTRATION ===");

        Thread demoThread = new Thread(() -> {
            System.out.println("Demo thread running: " + Thread.
               currentThread().getName());

            // Demonstrate sleep
            try {
                System.out.println("Demo thread: Going to sleep for 2
                   seconds");
```

```
246                 Thread.sleep(2000);
247                 System.out.println("Demo thread: Woke up from sleep");
248             } catch (InterruptedException e) {
249                 System.out.println("Demo thread: Sleep interrupted");
250             }
251 
252             // Demonstrate yield
253             System.out.println("Demo thread: Yielding to other threads"
                    );
254             Thread.yield();
255 
256             System.out.println("Demo thread: Continuing execution");
257 
258             // Self-interruption check
259             if (Thread.interrupted()) {
260                 System.out.println("Demo thread: Was interrupted");
261             }
262         }, "DemoThread");
263 
264         // Set thread properties
265         demoThread.setPriority(Thread.MAX_PRIORITY);
266 
267         System.out.println("Thread state before start: " + demoThread.
                getState());
268         System.out.println("Is Alive before start: " + demoThread.
                isAlive());
269 
270         demoThread.start();
271 
272         System.out.println("Thread state after start: " + demoThread.
                getState());
273         System.out.println("Is Alive after start: " + demoThread.
                isAlive());
274 
275         // Wait for thread to complete
276         try {
277             demoThread.join();
278             System.out.println("Thread state after join: " + demoThread
                    .getState());
279             System.out.println("Is Alive after join: " + demoThread.
                    isAlive());
280         } catch (InterruptedException e) {
281             e.printStackTrace();
282         }
283     }
284 
285     // ==================== MAIN METHOD ====================
286     public static void main(String[] args) {
287         System.out.println("=== CREATING THREADS BY EXTENDING THREAD
                CLASS ===\n");
288 
289         System.out.println("Main thread started: " + Thread.
                currentThread().getName());
290 
291         // 1. Basic Thread Example
292         System.out.println("\n=== 1. BASIC THREAD EXAMPLE ===");
293         SimpleThread thread1 = new SimpleThread("Thread-1");
294         SimpleThread thread2 = new SimpleThread("Thread-2");
```

```
295
296         thread1.start();
297         thread2.start();
298
299         // Wait for threads to complete
300         try {
301             thread1.join();
302             thread2.join();
303         } catch (InterruptedException e) {
304             e.printStackTrace();
305         }
306
307         // 2. Counter Thread Example
308         System.out.println("\n=== 2. COUNTER THREAD EXAMPLE ===");
309         CounterThread counter1 = new CounterThread("Counter-1",
                5000000);
310         CounterThread counter2 = new CounterThread("Counter-2",
                3000000);
311
312         // Set different priorities
313         counter1.setPriority(Thread.MIN_PRIORITY);
314         counter2.setPriority(Thread.MAX_PRIORITY);
315
316         counter1.start();
317         counter2.start();
318
319         // Interrupt one thread after some time
320         new Thread(() -> {
321             try {
322                 Thread.sleep(100);
323                 System.out.println("\nInterrupting Counter-1...");
324                 counter1.interrupt();
325             } catch (InterruptedException e) {
326                 e.printStackTrace();
327             }
328         }).start();
329
330         try {
331             counter1.join();
332             counter2.join();
333         } catch (InterruptedException e) {
334             e.printStackTrace();
335         }
336
337         // 3. Exception Handling Thread
338         System.out.println("\n=== 3. EXCEPTION HANDLING THREAD ===");
339         ExceptionHandlingThread exceptionThread = new
                ExceptionHandlingThread("Exception-Thread");
340         exceptionThread.start();
341
342         try {
343             exceptionThread.join();
344         } catch (InterruptedException e) {
345             e.printStackTrace();
346         }
347
348         // 4. Priority Demonstration
```

```
        System.out.println("\n=== 4. THREAD PRIORITY DEMONSTRATION ===");
        PriorityDemoThread lowPriority = new PriorityDemoThread("Low-Priority");
        PriorityDemoThread highPriority = new PriorityDemoThread("High-Priority");

        lowPriority.setPriority(Thread.MIN_PRIORITY);
        highPriority.setPriority(Thread.MAX_PRIORITY);

        lowPriority.start();
        highPriority.start();

        try {
            lowPriority.join();
            highPriority.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Results:");
        System.out.println("  " + lowPriority.getName() + ": " + lowPriority.getCount() +
                           " counts with priority " + lowPriority.getPriority());
        System.out.println("  " + highPriority.getName() + ": " + highPriority.getCount() +
                           " counts with priority " + highPriority.getPriority());

        // 5. Daemon Thread Example
        System.out.println("\n=== 5. DAEMON THREAD EXAMPLE ===");
        DaemonThread daemon = new DaemonThread("Background-Daemon");
        Thread userThread = new Thread(() -> {
            System.out.println("User thread started");
            try {
                Thread.sleep(3000); // User thread runs for 3 seconds
                System.out.println("User thread completed");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "User-Thread");

        daemon.start();
        userThread.start();

        try {
            userThread.join(); // Wait for user thread to complete
            System.out.println("Main thread: User thread completed, JVM may exit");
            System.out.println("Daemon thread state: " + daemon.getState());
            System.out.println("Daemon is alive: " + daemon.isAlive());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 6. Practical Download Manager
```

```
        System.out.println("\n=== 6. PRACTICAL DOWNLOAD MANAGER ===");
        DownloadThread download1 = new DownloadThread("file1.zip", 50);
        DownloadThread download2 = new DownloadThread("file2.iso", 30);

        download1.start();
        download2.start();

        // Control downloads
        new Thread(() -> {
            try {
                Thread.sleep(1500);
                download1.pauseDownload();

                Thread.sleep(1000);
                download1.resumeDownload();

                Thread.sleep(2000);
                download2.cancelDownload();

            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();

        try {
            download1.join();
            download2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 7. Thread Methods Demonstration
        demonstrateThreadMethods();

        System.out.println("\n=== KEY POINTS: EXTENDING THREAD CLASS
            ===");
        System.out.println("Advantages:");
        System.out.println("  1. Simple and straightforward");
        System.out.println("  2. Direct access to Thread class methods"
            );
        System.out.println("  3. Can override other Thread methods if
            needed");

        System.out.println("\nDisadvantages:");
        System.out.println("  1. Cannot extend other classes (Java
            single inheritance)");
        System.out.println("  2. Tight coupling with Thread class");
        System.out.println("  3. Less flexible than Runnable interface"
            );

        System.out.println("\nWhen to use:");
        System.out.println("  1. When you need to override Thread
            methods");
        System.out.println("  2. For simple thread implementations");
        System.out.println("  3. When you don't need to extend another
            class");

        System.out.println("\n=== THREAD CLASS IMPORTANT METHODS ===");
```

```
449             System.out.println("  - start(): Begins thread execution");
450             System.out.println("  - run(): Contains thread's code");
451             System.out.println("  - sleep(): Pauses thread for specified
                    time");
452             System.out.println("  - join(): Waits for thread to die");
453             System.out.println("  - yield(): Hints scheduler to give up CPU
                    ");
454             System.out.println("  - interrupt(): Interrupts the thread");
455             System.out.println("  - isAlive(): Checks if thread is alive");
456             System.out.println("  - setPriority(): Sets thread priority");
457             System.out.println("  - setDaemon(): Marks as daemon thread");
458
459             System.out.println("\nMain thread completed");
460         }
461 }
```

Listing 2: Creating Threads by Extending Thread Class

ExtendingThreadClass Program Output

```
=== CREATING THREADS BY EXTENDING THREAD CLASS ===

Main thread started: main

=== 1. BASIC THREAD EXAMPLE ===
Creating thread: Thread-1
Creating thread: Thread-2
Thread Thread-1 is running
Thread-1: Count 1
Thread Thread-2 is running
Thread-2: Count 1
Thread-1: Count 2
Thread-2: Count 2
Thread-1: Count 3
Thread-2: Count 3
Thread-1: Count 4
Thread-2: Count 4
Thread-1: Count 5
Thread-2: Count 5
Thread Thread-2 exiting
Thread Thread-1 exiting

=== 2. COUNTER THREAD EXAMPLE ===
Counter-1 started with priority: 1
Counter-2 started with priority: 10
Counter-2: Processed 1000000 numbers
Counter-1: Processed 1000000 numbers
Counter-2: Processed 2000000 numbers

Interrupting Counter-1...
Counter-1: Interruption detected, cleaning up...
```

```
Counter-1: Sum of first 1000000 numbers = 1784293664
Counter-1 completed
Counter-2: Processed 3000000 numbers
Counter-2: Yielding to other threads
Counter-2: Sum of first 3000000 numbers = 4500001500000
Counter-2 completed

=== 3. EXCEPTION HANDLING THREAD ===
Exception-Thread started
Exception-Thread: Working on step 1
Exception-Thread: Working on step 2
Exception-Thread: Working on step 3
Exception-Thread: Working on step 4
Exception-Thread: Working on step 5
Exception-Thread: Working on step 6
Exception-Thread caught exception: Simulated error in Exception-Thread
Exception-Thread in finally block - cleaning up
Exception-Thread completed execution

=== 4. THREAD PRIORITY DEMONSTRATION ===
Low-Priority started with priority 1
High-Priority started with priority 10
High-Priority completed. Count: 892, Priority: 10
Low-Priority completed. Count: 267, Priority: 1
Results:
  Low-Priority: 267 counts with priority 1
  High-Priority: 892 counts with priority 10

=== 5. DAEMON THREAD EXAMPLE ===
Background-Daemon started. Is Daemon: true
Background-Daemon: Background task 1
User thread started
Background-Daemon: Background task 2
Background-Daemon: Background task 3
User thread completed
Main thread: User thread completed, JVM may exit
Daemon thread state: TIMED_WAITING
Daemon is alive: true

=== 6. PRACTICAL DOWNLOAD MANAGER ===
Download-file1.zip: Starting download of file1.zip (50 MB)
Download-file2.iso: Starting download of file2.iso (30 MB)
Download-file1.zip: 20% completed (10/50 MB)
Download-file2.iso: 33% completed (10/30 MB)
Download-file1.zip: Download paused
Download-file2.iso: 66% completed (20/30 MB)
Download-file1.zip: Download resumed
```

```
Download-file2.iso: Download cancellation requested
Download-file2.iso: Download cancelled
Download-file1.zip: 40% completed (20/50 MB)
Download-file1.zip: 60% completed (30/50 MB)
Download-file1.zip: 80% completed (40/50 MB)
Download-file1.zip: 100% completed (50/50 MB)
Download-file1.zip: Download completed successfully!

=== THREAD METHODS DEMONSTRATION ===
Thread state before start: NEW
Is Alive before start: false
Thread state after start: RUNNABLE
Is Alive after start: true
Demo thread running: DemoThread
Demo thread: Going to sleep for 2 seconds
Demo thread: Woke up from sleep
Demo thread: Yielding to other threads
Demo thread: Continuing execution
Thread state after join: TERMINATED
Is Alive after join: false

=== KEY POINTS: EXTENDING THREAD CLASS ===
Advantages:
  1. Simple and straightforward
  2. Direct access to Thread class methods
  3. Can override other Thread methods if needed

Disadvantages:
  1. Cannot extend other classes (Java single inheritance)
  2. Tight coupling with Thread class
  3. Less flexible than Runnable interface

When to use:
  1. When you need to override Thread methods
  2. For simple thread implementations
  3. When you don't need to extend another class

=== THREAD CLASS IMPORTANT METHODS ===
  - start(): Begins thread execution
  - run(): Contains thread's code
  - sleep(): Pauses thread for specified time
  - join(): Waits for thread to die
  - yield(): Hints scheduler to give up CPU
  - interrupt(): Interrupts the thread
  - isAlive(): Checks if thread is alive
  - setPriority(): Sets thread priority
  - setDaemon(): Marks as daemon thread
```

```
Main thread completed
```

3.2 Implementing Runnable Interface

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

public class ImplementingRunnableInterface {

    // ==================== BASIC RUNNABLE IMPLEMENTATION ====================
    static class SimpleRunnable implements Runnable {
        private final String taskName;

        public SimpleRunnable(String name) {
            this.taskName = name;
        }

        @Override
        public void run() {
            System.out.println(taskName + " started in thread: " +
                               Thread.currentThread().getName());

            try {
                // Simulate work
                for (int i = 1; i <= 5; i++) {
                    System.out.println(taskName + ": Processing item " + i);
                    Thread.sleep(300);
                }
            } catch (InterruptedException e) {
                System.out.println(taskName + " was interrupted");
            }

            System.out.println(taskName + " completed");
        }
    }

    // ==================== RUNNABLE WITH RETURN VALUE (USING FUTURE) ====================
    static class CalculationRunnable implements Runnable {
        private final int number;
        private int result;

        public CalculationRunnable(int number) {
            this.number = number;
        }

        @Override
        public void run() {
            System.out.println("Calculating factorial of " + number +
                               public int getResult() {
            return result;
        }
    }
```

```
48
49      // ==================== CALLABLE FOR RETURN VALUE ====================
50      static class FactorialCallable implements Callable<Long> {
51          private final int number;
52
53          public FactorialCallable(int number) {
54              this.number = number;
55          }
56
57          @Override
58          public Long call() throws Exception {
59              System.out.println("Calculating factorial of " + number +
60                                 " in thread: " + Thread.currentThread().
                                       getName());
61
62              long factorial = 1;
63              for (int i = 2; i <= number; i++) {
64                  factorial *= i;
65                  Thread.sleep(10); // Simulate computation time
66              }
67
68              return factorial;
69          }
70      }
71
72      // ==================== RUNNABLE WITH SHARED RESOURCES ====================
73      static class BankAccount {
74          private int balance;
75
76          public BankAccount(int initialBalance) {
77              this.balance = initialBalance;
78          }
79
80          public synchronized void deposit(int amount) {
81              balance += amount;
82              System.out.println(Thread.currentThread().getName() +
83                                 ": Deposited " + amount + ", Balance: " +
                                       balance);
84          }
85
86          public synchronized void withdraw(int amount) {
87              if (balance >= amount) {
88                  balance -= amount;
89                  System.out.println(Thread.currentThread().getName() +
90                                     ": Withdrawn " + amount + ", Balance:
                                           " + balance);
91              } else {
92                  System.out.println(Thread.currentThread().getName() +
93                                     ": Insufficient funds for withdrawal
                                           of " + amount);
94              }
95          }
96
97          public int getBalance() {
98              return balance;
99          }
```

```
100     }
101
102     static class TransactionRunnable implements Runnable {
103         private final BankAccount account;
104         private final boolean isDeposit;
105         private final int amount;
106
107         public TransactionRunnable(BankAccount account, boolean
                isDeposit, int amount) {
108             this.account = account;
109             this.isDeposit = isDeposit;
110             this.amount = amount;
111         }
112
113         @Override
114         public void run() {
115             if (isDeposit) {
116                 account.deposit(amount);
117             } else {
118                 account.withdraw(amount);
119             }
120         }
121     }
122
123     // ==================== RUNNABLE WITH ATOMIC OPERATIONS
            ====================
124     static class AtomicCounterRunnable implements Runnable {
125         private static final AtomicInteger globalCounter = new
                AtomicInteger(0);
126         private final String workerName;
127         private final int iterations;
128
129         public AtomicCounterRunnable(String name, int iterations) {
130             this.workerName = name;
131             this.iterations = iterations;
132         }
133
134         @Override
135         public void run() {
136             System.out.println(workerName + " started");
137
138             for (int i = 0; i < iterations; i++) {
139                 int currentValue = globalCounter.incrementAndGet();
140
141                 if (i % 100000 == 0) {
142                     System.out.println(workerName + ": Incremented to "
                            + currentValue);
143                 }
144
145                 // Simulate some work
146                 for (int j = 0; j < 100; j++) {
147                     // Busy wait
148                 }
149             }
150
151             System.out.println(workerName + " completed. Final count: "
                    +
152                                globalCounter.get());
```

```
153         }
154     }
155
156     // ==================== THREAD POOL WITH RUNNABLE
        ====================
157     static class ThreadPoolWorker implements Runnable {
158         private final int taskId;
159         private final int duration;
160
161         public ThreadPoolWorker(int taskId, int duration) {
162             this.taskId = taskId;
163             this.duration = duration;
164         }
165
166         @Override
167         public void run() {
168             System.out.println("Task " + taskId + " started by: " +
169                                Thread.currentThread().getName());
170
171             try {
172                 Thread.sleep(duration); // Simulate work
173                 System.out.println("Task " + taskId + " completed by: "
                        +
174                                    Thread.currentThread().getName());
175             } catch (InterruptedException e) {
176                 System.out.println("Task " + taskId + " interrupted");
177             }
178         }
179     }
180
181     // ==================== RUNNABLE WITH COMPLEX STATE
        ====================
182     static class SensorDataProcessor implements Runnable {
183         private final String sensorId;
184         private volatile boolean running = true;
185         private int dataPointsProcessed = 0;
186
187         public SensorDataProcessor(String sensorId) {
188             this.sensorId = sensorId;
189         }
190
191         @Override
192         public void run() {
193             System.out.println(sensorId + " data processor started");
194
195             try {
196                 while (running && !Thread.currentThread().isInterrupted
                        ()) {
197                     // Simulate data processing
198                     processData();
199                     dataPointsProcessed++;
200
201                     if (dataPointsProcessed % 10 == 0) {
202                         System.out.println(sensorId + ": Processed " +
203                                            dataPointsProcessed + " data
                                               points");
204                     }
205
```

```
                    Thread.sleep(100); // Process data every 100ms
                }
            } catch (InterruptedException e) {
                System.out.println(sensorId + " interrupted");
            } finally {
                System.out.println(sensorId + " shutting down. Total
                    processed: " +
                                   dataPointsProcessed);
            }
        }

        private void processData() {
            // Simulate data processing
            double value = Math.random() * 100;
            // In real application, this would process actual sensor
                data
        }

        public void stopProcessing() {
            running = false;
        }

        public int getDataPointsProcessed() {
            return dataPointsProcessed;
        }
    }

    // ==================== RUNNABLE WITH PARAMETERS AND RESULTS
        ====================
    static class MatrixMultiplier implements Runnable {
        private final int[][] matrixA;
        private final int[][] matrixB;
        private final int[][] result;
        private final int rowStart;
        private final int rowEnd;

        public MatrixMultiplier(int[][] matrixA, int[][] matrixB,
                                int[][] result, int rowStart, int rowEnd
                                    ) {
            this.matrixA = matrixA;
            this.matrixB = matrixB;
            this.result = result;
            this.rowStart = rowStart;
            this.rowEnd = rowEnd;
        }

        @Override
        public void run() {
            int colsB = matrixB[0].length;
            int colsA = matrixA[0].length;

            for (int i = rowStart; i < rowEnd; i++) {
                for (int j = 0; j < colsB; j++) {
                    result[i][j] = 0;
                    for (int k = 0; k < colsA; k++) {
                        result[i][j] += matrixA[i][k] * matrixB[k][j];
                    }
                }
```

```
            }

            System.out.println(Thread.currentThread().getName() +
                               ": Processed rows " + rowStart + " to " +
                                   (rowEnd - 1));
        }
    }

    // ==================== DEMONSTRATION METHODS ====================

    public static void demonstrateBasicRunnable() {
        System.out.println("\n=== BASIC RUNNABLE DEMONSTRATION ===");

        // Create Runnable tasks
        Runnable task1 = new SimpleRunnable("Task-1");
        Runnable task2 = new SimpleRunnable("Task-2");
        Runnable task3 = () -> {
            System.out.println("Lambda Runnable in thread: " +
                               Thread.currentThread().getName());
            try {
                Thread.sleep(500);
                System.out.println("Lambda Runnable completed");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        };

        // Create threads for each runnable
        Thread thread1 = new Thread(task1, "Worker-1");
        Thread thread2 = new Thread(task2, "Worker-2");
        Thread thread3 = new Thread(task3, "Worker-3");

        // Start threads
        thread1.start();
        thread2.start();
        thread3.start();

        // Wait for completion
        try {
            thread1.join();
            thread2.join();
            thread3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void demonstrateThreadPool() throws Exception {
        System.out.println("\n=== THREAD POOL DEMONSTRATION ===");

        // Create thread pool
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Submit tasks
        for (int i = 1; i <= 10; i++) {
            int duration = (int) (Math.random() * 1000) + 500;
            executor.execute(new ThreadPoolWorker(i, duration));
        }
```

```
317
318         // Shutdown executor
319         executor.shutdown();
320         executor.awaitTermination(10, TimeUnit.SECONDS);
321
322         System.out.println("All tasks completed in thread pool");
323     }
324
325     public static void demonstrateCallableFuture() throws Exception {
326         System.out.println("\n=== CALLABLE AND FUTURE DEMONSTRATION ===
                ");
327
328         ExecutorService executor = Executors.newFixedThreadPool(3);
329
330         // Submit Callable tasks
331         Future<Long> future1 = executor.submit(new FactorialCallable
                (10));
332         Future<Long> future2 = executor.submit(new FactorialCallable
                (15));
333         Future<Long> future3 = executor.submit(new FactorialCallable
                (20));
334
335         // Get results (blocks until available)
336         System.out.println("Factorial of 10 = " + future1.get());
337         System.out.println("Factorial of 15 = " + future2.get());
338         System.out.println("Factorial of 20 = " + future3.get());
339
340         executor.shutdown();
341     }
342
343     public static void demonstrateBankAccount() {
344         System.out.println("\n=== BANK ACCOUNT CONCURRENCY
                DEMONSTRATION ===");
345
346         BankAccount account = new BankAccount(1000);
347
348         // Create multiple transaction threads
349         Thread[] threads = new Thread[10];
350         for (int i = 0; i < 10; i++) {
351             boolean isDeposit = Math.random() > 0.5;
352             int amount = (int) (Math.random() * 200) + 50;
353             threads[i] = new Thread(
354                 new TransactionRunnable(account, isDeposit, amount),
355                 "Transaction-" + (i + 1)
356             );
357         }
358
359         // Start all threads
360         for (Thread thread : threads) {
361             thread.start();
362         }
363
364         // Wait for completion
365         for (Thread thread : threads) {
366             try {
367                 thread.join();
368             } catch (InterruptedException e) {
369                 e.printStackTrace();
```

```
            }
        }

        System.out.println("Final balance: " + account.getBalance());
    }

    public static void demonstrateMatrixMultiplication() {
        System.out.println("\n=== PARALLEL MATRIX MULTIPLICATION ===");

        int size = 100;
        int[][] matrixA = new int[size][size];
        int[][] matrixB = new int[size][size];
        int[][] result = new int[size][size];

        // Initialize matrices with random values
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                matrixA[i][j] = (int) (Math.random() * 10);
                matrixB[i][j] = (int) (Math.random() * 10);
            }
        }

        int numThreads = 4;
        int rowsPerThread = size / numThreads;
        Thread[] threads = new Thread[numThreads];

        long startTime = System.currentTimeMillis();

        // Create and start threads
        for (int i = 0; i < numThreads; i++) {
            int rowStart = i * rowsPerThread;
            int rowEnd = (i == numThreads - 1) ? size : rowStart +
                rowsPerThread;

            threads[i] = new Thread(
                new MatrixMultiplier(matrixA, matrixB, result, rowStart
                    , rowEnd),
                "MatrixWorker-" + i
            );
            threads[i].start();
        }

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

        long endTime = System.currentTimeMillis();

        System.out.println("Matrix multiplication completed in " +
                           (endTime - startTime) + "ms");

        // Verify a sample result
        System.out.println("Sample result[0][0] = " + result[0][0]);
```

```
426     }
427
428     // ==================== MAIN METHOD ====================
429     public static void main(String[] args) throws Exception {
430         System.out.println("=== CREATING THREADS BY IMPLEMENTING
                RUNNABLE INTERFACE ===\n");
431
432         System.out.println("Main thread: " + Thread.currentThread().
                getName());
433
434         // 1. Basic Runnable demonstration
435         demonstrateBasicRunnable();
436
437         // 2. Thread Pool demonstration
438         demonstrateThreadPool();
439
440         // 3. Callable and Future demonstration
441         demonstrateCallableFuture();
442
443         // 4. Bank account concurrency demonstration
444         demonstrateBankAccount();
445
446         // 5. Atomic operations demonstration
447         System.out.println("\n=== ATOMIC OPERATIONS DEMONSTRATION ===")
                ;
448         Thread atomic1 = new Thread(new AtomicCounterRunnable("Atomic-1
                ", 1000000));
449         Thread atomic2 = new Thread(new AtomicCounterRunnable("Atomic-2
                ", 1000000));
450         Thread atomic3 = new Thread(new AtomicCounterRunnable("Atomic-3
                ", 1000000));
451
452         atomic1.start();
453         atomic2.start();
454         atomic3.start();
455
456         atomic1.join();
457         atomic2.join();
458         atomic3.join();
459
460         // 6. Sensor data processor demonstration
461         System.out.println("\n=== SENSOR DATA PROCESSOR DEMONSTRATION
                ===");
462         SensorDataProcessor tempSensor = new SensorDataProcessor("
                Temperature");
463         SensorDataProcessor pressureSensor = new SensorDataProcessor("
                Pressure");
464
465         Thread sensorThread1 = new Thread(tempSensor, "Temp-Processor")
                ;
466         Thread sensorThread2 = new Thread(pressureSensor, "Pressure-
                Processor");
467
468         sensorThread1.start();
469         sensorThread2.start();
470
471         // Let them run for 3 seconds
472         Thread.sleep(3000);
```

```
473
474         // Stop processing
475         tempSensor.stopProcessing();
476         pressureSensor.stopProcessing();
477
478         sensorThread1.interrupt();
479         sensorThread2.interrupt();
480
481         sensorThread1.join();
482         sensorThread2.join();
483
484         // 7. Parallel matrix multiplication
485         demonstrateMatrixMultiplication();
486
487         System.out.println("\n=== ADVANTAGES OF RUNNABLE INTERFACE ===");
488         System.out.println("1. Flexibility: Can extend another class");
489         System.out.println("2. Reusability: Same Runnable can be used with different threads");
490         System.out.println("3. Thread Pool Support: Works well with ExecutorService");
491         System.out.println("4. Lambda Support: Can be implemented with lambdas");
492         System.out.println("5. Better Design: Separates task from execution mechanism");
493
494         System.out.println("\n=== RUNNABLE VS THREAD EXTENSION ===");
495         System.out.println("Runnable Interface:");
496         System.out.println("  - Better for code reuse");
497         System.out.println("  - Allows extending other classes");
498         System.out.println("  - Separates task from thread mechanics");
499         System.out.println("  - Recommended for most cases");
500
501         System.out.println("\nThread Extension:");
502         System.out.println("  - Simpler for basic cases");
503         System.out.println("  - Direct access to Thread methods");
504         System.out.println("  - Cannot extend other classes");
505         System.out.println("  - Use when you need to override Thread behavior");
506
507         System.out.println("\n=== EXECUTOR SERVICE BENEFITS ===");
508         System.out.println("1. Thread Reuse: Reduces thread creation overhead");
509         System.out.println("2. Resource Management: Controls number of concurrent threads");
510         System.out.println("3. Task Queueing: Handles tasks when all threads are busy");
511         System.out.println("4. Future Results: Callable tasks can return results");
512         System.out.println("5. Scheduled Execution: Can schedule tasks for future");
513
514         System.out.println("\n=== BEST PRACTICES ===");
515         System.out.println("1. Prefer Runnable over Thread extension");
516         System.out.println("2. Use ExecutorService for thread management");
517         System.out.println("3. Implement Callable when you need return values");
```

```
518             System.out.println("4. Use atomic variables for thread-safe
                    counters");
519             System.out.println("5. Always handle InterruptedException
                    properly");
520
521             System.out.println("\nMain thread completed");
522         }
523 }
```

Listing 3: Creating Threads by Implementing Runnable Interface

ImplementingRunnableInterface Program Output

```
=== CREATING THREADS BY IMPLEMENTING RUNNABLE INTERFACE ===

Main thread: main

=== BASIC RUNNABLE DEMONSTRATION ===
Task-1 started in thread: Worker-1
Task-1: Processing item 1
Task-2 started in thread: Worker-2
Task-2: Processing item 1
Lambda Runnable in thread: Worker-3
Task-1: Processing item 2
Task-2: Processing item 2
Task-1: Processing item 3
Task-2: Processing item 3
Lambda Runnable completed
Task-1: Processing item 4
Task-2: Processing item 4
Task-1: Processing item 5
Task-2: Processing item 5
Task-1 completed
Task-2 completed

=== THREAD POOL DEMONSTRATION ===
Task 1 started by: pool-1-thread-1
Task 2 started by: pool-1-thread-2
Task 3 started by: pool-1-thread-3
Task 4 started by: pool-1-thread-2
Task 1 completed by: pool-1-thread-1
Task 5 started by: pool-1-thread-1
Task 2 completed by: pool-1-thread-2
Task 6 started by: pool-1-thread-2
Task 3 completed by: pool-1-thread-3
Task 7 started by: pool-1-thread-3
Task 5 completed by: pool-1-thread-1
Task 8 started by: pool-1-thread-1
Task 4 completed by: pool-1-thread-2
```

```
Task 9 started by: pool-1-thread-2
Task 6 completed by: pool-1-thread-2
Task 10 started by: pool-1-thread-2
Task 7 completed by: pool-1-thread-3
Task 8 completed by: pool-1-thread-1
Task 9 completed by: pool-1-thread-2
Task 10 completed by: pool-1-thread-2
All tasks completed in thread pool

=== CALLABLE AND FUTURE DEMONSTRATION ===
Calculating factorial of 10 in thread: pool-2-thread-1
Calculating factorial of 15 in thread: pool-2-thread-2
Calculating factorial of 20 in thread: pool-2-thread-3
Factorial of 10 = 3628800
Factorial of 15 = 1307674368000
Factorial of 20 = 2432902008176640000

=== BANK ACCOUNT CONCURRENCY DEMONSTRATION ===
Transaction-1: Deposited 197, Balance: 1197
Transaction-2: Withdrawn 162, Balance: 1035
Transaction-3: Withdrawn 221, Balance: 814
Transaction-4: Withdrawn 73, Balance: 741
Transaction-5: Deposited 93, Balance: 834
Transaction-6: Withdrawn 244, Balance: 590
Transaction-7: Deposited 114, Balance: 704
Transaction-8: Deposited 248, Balance: 952
Transaction-9: Deposited 218, Balance: 1170
Transaction-10: Withdrawn 237, Balance: 933
Final balance: 933

=== ATOMIC OPERATIONS DEMONSTRATION ===
Atomic-1 started
Atomic-2 started
Atomic-3 started
Atomic-1: Incremented to 1
Atomic-2: Incremented to 2
Atomic-3: Incremented to 3
Atomic-1: Incremented to 100001
Atomic-2: Incremented to 200001
Atomic-3: Incremented to 300001
Atomic-1: Incremented to 200001
Atomic-2: Incremented to 400001
Atomic-3: Incremented to 600001
Atomic-1: Incremented to 300001
Atomic-2: Incremented to 600001
Atomic-3: Incremented to 900001
Atomic-1 completed. Final count: 1000000
```

```
Atomic-2 completed. Final count: 2000000
Atomic-3 completed. Final count: 3000000

=== SENSOR DATA PROCESSOR DEMONSTRATION ===
Temperature data processor started
Pressure data processor started
Temperature: Processed 10 data points
Pressure: Processed 10 data points
Temperature: Processed 20 data points
Pressure: Processed 20 data points
Temperature: Processed 30 data points
Pressure: Processed 30 data points
Temperature interrupted
Temperature shutting down. Total processed: 30
Pressure interrupted
Pressure shutting down. Total processed: 30

=== PARALLEL MATRIX MULTIPLICATION ===
MatrixWorker-0: Processed rows 0 to 24
MatrixWorker-2: Processed rows 50 to 74
MatrixWorker-1: Processed rows 25 to 49
MatrixWorker-3: Processed rows 75 to 99
Matrix multiplication completed in 15ms
Sample result[0][0] = 2347

=== ADVANTAGES OF RUNNABLE INTERFACE ===
1. Flexibility: Can extend another class
2. Reusability: Same Runnable can be used with different threads
3. Thread Pool Support: Works well with ExecutorService
4. Lambda Support: Can be implemented with lambdas
5. Better Design: Separates task from execution mechanism

=== RUNNABLE VS THREAD EXTENSION ===
Runnable Interface:
  - Better for code reuse
  - Allows extending other classes
  - Separates task from thread mechanics
  - Recommended for most cases

Thread Extension:
  - Simpler for basic cases
  - Direct access to Thread methods
  - Cannot extend other classes
  - Use when you need to override Thread behavior

=== EXECUTOR SERVICE BENEFITS ===
1. Thread Reuse: Reduces thread creation overhead
```

```
2. Resource Management: Controls number of concurrent threads
3. Task Queueing: Handles tasks when all threads are busy
4. Future Results: Callable tasks can return results
5. Scheduled Execution: Can schedule tasks for future

=== BEST PRACTICES ===
1. Prefer Runnable over Thread extension
2. Use ExecutorService for thread management
3. Implement Callable when you need return values
4. Use atomic variables for thread-safe counters
5. Always handle InterruptedException properly

Main thread completed
```

4 Thread Life Cycle and States

4.1 Thread States in Java

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

public class ThreadLifeCycle {

    // ==================== THREAD STATE MONITOR ====================
    static class ThreadStateMonitor implements Runnable {
        private final Thread targetThread;
        private volatile boolean monitoring = true;

        public ThreadStateMonitor(Thread targetThread) {
            this.targetThread = targetThread;
        }

        @Override
        public void run() {
            System.out.println("Starting thread state monitor for: " +
                               targetThread.getName());

            while (monitoring && !Thread.currentThread().isInterrupted
                ()) {
                Thread.State state = targetThread.getState();
                System.out.println("[" + System.currentTimeMillis() + "
                    ] " +
                                   targetThread.getName() + " state: " +
                                       state);

                try {
                    Thread.sleep(100); // Monitor every 100ms
                } catch (InterruptedException e) {
                    System.out.println("Monitor interrupted");
                    break;
                }
            }
```

```
32
33              System.out.println("Thread state monitor stopped");
34          }
35
36          public void stopMonitoring() {
37              monitoring = false;
38          }
39      }
40
41      // ==================== THREAD WITH ALL STATES ====================
42      static class StateDemonstrationThread extends Thread {
43          private final Lock lock = new ReentrantLock();
44          private volatile boolean conditionMet = false;
45
46          public StateDemonstrationThread(String name) {
47              super(name);
48          }
49
50          @Override
51          public void run() {
52              System.out.println(getName() + ": Starting execution -
                    State: " + getState());
53
54              // 1. RUNNABLE state
55              System.out.println(getName() + ": Entering RUNNABLE state")
                    ;
56              performComputation();
57
58              // 2. TIMED_WAITING state (sleep)
59              System.out.println(getName() + ": Entering TIMED_WAITING
                    state (sleep)");
60              try {
61                  Thread.sleep(2000);
62              } catch (InterruptedException e) {
63                  System.out.println(getName() + ": Sleep interrupted");
64                  return;
65              }
66
67              // 3. WAITING state (wait on monitor)
68              System.out.println(getName() + ": Entering WAITING state (
                    wait)");
69              synchronized (this) {
70                  try {
71                      while (!conditionMet) {
72                          wait(); // Releases lock and waits
73                      }
74                  } catch (InterruptedException e) {
75                      System.out.println(getName() + ": Wait interrupted"
                            );
76                      return;
77                  }
78              }
79
80              // 4. BLOCKED state (waiting for lock)
81              System.out.println(getName() + ": Attempting to acquire
                    lock - may enter BLOCKED");
82              lock.lock();
83              try {
```

```
84                  System.out.println(getName() + ": Acquired lock, doing
                        work...");
85                  Thread.sleep(500);
86              } catch (InterruptedException e) {
87                  System.out.println(getName() + ": Interrupted while
                        holding lock");
88              } finally {
89                  lock.unlock();
90              }
91
92              // 5. TERMINATED state (implicitly after run() completes)
93              System.out.println(getName() + ": Completing execution -
                    will enter TERMINATED");
94          }
95
96          private void performComputation() {
97              long sum = 0;
98              for (int i = 0; i < 1000000; i++) {
99                  sum += i;
100             }
101             System.out.println(getName() + ": Computation completed,
                    sum = " + sum);
102         }
103
104         public synchronized void notifyCondition() {
105             conditionMet = true;
106             notify(); // Bring out of WAITING state
107         }
108     }
109
110     // ==================== THREAD STATE TRANSITION DEMO
            ====================
111     public static void demonstrateStateTransitions() throws Exception {
112         System.out.println("\n=== THREAD STATE TRANSITION DEMONSTRATION
                ===");
113
114         // Create demonstration thread
115         StateDemonstrationThread demoThread = new
                StateDemonstrationThread("DemoThread");
116
117         // Create and start monitor
118         ThreadStateMonitor monitor = new ThreadStateMonitor(demoThread)
                ;
119         Thread monitorThread = new Thread(monitor, "MonitorThread");
120         monitorThread.setDaemon(true);
121         monitorThread.start();
122
123         // Start the demonstration thread
124         System.out.println("Initial state: " + demoThread.getState());
                // NEW
125
126         demoThread.start(); // Transition to RUNNABLE
127         Thread.sleep(50);
128
129         // Let thread sleep (TIMED_WAITING)
130         Thread.sleep(100);
131         System.out.println("\nThread is sleeping (TIMED_WAITING)...");
132
```

```
133         // Wait for thread to enter WAITING state
134         Thread.sleep(2200);
135 
136         // Create another thread to acquire lock first (to cause
                BLOCKED state)
137         Lock sharedLock = new ReentrantLock();
138         Thread blockingThread = new Thread(() -> {
139             System.out.println("BlockingThread: Acquiring lock first...
                    ");
140             sharedLock.lock();
141             try {
142                 Thread.sleep(1000); // Hold lock for 1 second
143             } catch (InterruptedException e) {
144                 e.printStackTrace();
145             } finally {
146                 sharedLock.unlock();
147                 System.out.println("BlockingThread: Released lock");
148             }
149         }, "BlockingThread");
150 
151         // Notify waiting thread
152         System.out.println("\nNotifying waiting thread...");
153         demoThread.notifyCondition();
154 
155         // Start blocking thread
156         blockingThread.start();
157         Thread.sleep(100); // Give blocking thread time to acquire lock
158 
159         // Wait for termination
160         demoThread.join();
161         System.out.println("\nFinal state: " + demoThread.getState());
                // TERMINATED
162 
163         // Stop monitor
164         monitor.stopMonitoring();
165         Thread.sleep(200);
166     }
167 
168     // ==================== THREAD JOIN STATES ====================
169     static class JoiningThread extends Thread {
170         private final String workerName;
171         private final int workDuration;
172 
173         public JoiningThread(String name, int duration) {
174             this.workerName = name;
175             this.workDuration = duration;
176         }
177 
178         @Override
179         public void run() {
180             System.out.println(workerName + ": Starting work for " +
                    workDuration + "ms");
181 
182             try {
183                 Thread.sleep(workDuration);
184             } catch (InterruptedException e) {
185                 System.out.println(workerName + ": Interrupted");
186                 return;
```

```
            }

            System.out.println(workerName + ": Work completed");
        }
    }

    public static void demonstrateJoinStates() throws Exception {
        System.out.println("\n=== THREAD JOIN STATE DEMONSTRATION ===");

        JoiningThread worker1 = new JoiningThread("Worker-1", 2000);
        JoiningThread worker2 = new JoiningThread("Worker-2", 3000);

        System.out.println("Main thread state before starting workers: " +
                           Thread.currentThread().getState());

        worker1.start();
        worker2.start();

        System.out.println("\nWorker-1 state after start: " + worker1.getState());
        System.out.println("Worker-2 state after start: " + worker2.getState());

        // Create a thread to monitor main thread during join
        Thread monitorMain = new Thread(() -> {
            try {
                Thread.sleep(500);
                System.out.println("\n[Monitor] Main thread state during join: " +
                                   Thread.currentThread().getState());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "MainMonitor");
        monitorMain.setDaemon(true);
        monitorMain.start();

        System.out.println("\nMain thread: Waiting for Worker-1 to complete...");
        worker1.join(); // Main thread enters WAITING state
        System.out.println("Main thread: Worker-1 completed");

        System.out.println("Worker-1 state after join: " + worker1.getState());
        System.out.println("Worker-2 state while running: " + worker2.getState());

        System.out.println("\nMain thread: Waiting for Worker-2 to complete...");
        worker2.join(1000); // Timed wait - returns after timeout if thread not done
        System.out.println("Main thread: Join completed or timed out");

        if (worker2.isAlive()) {
            System.out.println("Worker-2 is still alive, interrupting...");
```

```
234             worker2.interrupt();
235             worker2.join();
236         }
237 
238         System.out.println("\nFinal states:");
239         System.out.println("Worker-1: " + worker1.getState());
240         System.out.println("Worker-2: " + worker2.getState());
241         System.out.println("Main thread: " + Thread.currentThread().
                getState());
242     }
243 
244     // ==================== THREAD INTERRUPTION STATES
            ====================
245     static class InterruptibleThread extends Thread {
246         public InterruptibleThread(String name) {
247             super(name);
248         }
249 
250         @Override
251         public void run() {
252             System.out.println(getName() + ": Starting, interrupt
                    status: " +
253                                isInterrupted());
254 
255             // Method 1: Check interrupt flag periodically
256             for (int i = 1; i <= 10; i++) {
257                 if (Thread.interrupted()) { // Clears interrupt status
258                     System.out.println(getName() + ": Interrupted
                            during work, cleaning up");
259                     return;
260                 }
261 
262                 System.out.println(getName() + ": Working on step " + i
                        );
263 
264                 try {
265                     Thread.sleep(500);
266                 } catch (InterruptedException e) {
267                     System.out.println(getName() + ": Sleep interrupted
                            , interrupt status: " +
268                                        isInterrupted());
269                     // Restore interrupt status
270                     Thread.currentThread().interrupt();
271                     return;
272                 }
273             }
274 
275             System.out.println(getName() + ": Completed normally");
276         }
277     }
278 
279     public static void demonstrateInterruptionStates() throws Exception
            {
280         System.out.println("\n=== THREAD INTERRUPTION STATES ===");
281 
282         InterruptibleThread thread1 = new InterruptibleThread("
                Interruptible-1");
```

```
283         InterruptibleThread thread2 = new InterruptibleThread("
                Interruptible-2");
284
285         thread1.start();
286         thread2.start();
287
288         // Interrupt thread1 while it's running
289         Thread.sleep(1200);
290         System.out.println("\nInterrupting thread1...");
291         thread1.interrupt();
292
293         // Interrupt thread2 while it's sleeping
294         Thread.sleep(100);
295         System.out.println("Interrupting thread2...");
296         thread2.interrupt();
297
298         thread1.join();
299         thread2.join();
300
301         System.out.println("\nFinal interrupt status:");
302         System.out.println("thread1: " + thread1.isInterrupted());
303         System.out.println("thread2: " + thread2.isInterrupted());
304     }
305
306     // ==================== DAEMON THREAD STATES ====================
307     public static void demonstrateDaemonStates() throws Exception {
308         System.out.println("\n=== DAEMON THREAD STATES ===");
309
310         Thread daemonThread = new Thread(() -> {
311             System.out.println("Daemon thread started, state: " +
312                                Thread.currentThread().getState());
313             System.out.println("Is Daemon: " + Thread.currentThread().
                    isDaemon());
314
315             try {
316                 // Daemon thread will be terminated when all user
                        threads finish
317                 for (int i = 1; i <= 10; i++) {
318                     System.out.println("Daemon: Working " + i);
319                     Thread.sleep(500);
320                 }
321             } catch (InterruptedException e) {
322                 System.out.println("Daemon interrupted");
323             }
324
325             System.out.println("Daemon completed (may not reach this)")
                    ;
326         }, "DaemonWorker");
327
328         daemonThread.setDaemon(true);
329
330         System.out.println("Daemon state before start: " + daemonThread
                .getState());
331         daemonThread.start();
332         System.out.println("Daemon state after start: " + daemonThread.
                getState());
333
334         // User thread that runs briefly
```

```
335         Thread userThread = new Thread(() -> {
336             try {
337                 System.out.println("User thread running for 2 seconds")
                        ;
338                 Thread.sleep(2000);
339                 System.out.println("User thread completed");
340             } catch (InterruptedException e) {
341                 e.printStackTrace();
342             }
343         }, "UserWorker");
344 
345         userThread.start();
346         userThread.join();
347 
348         System.out.println("\nAfter user thread completion:");
349         System.out.println("Daemon state: " + daemonThread.getState());
350         System.out.println("Daemon is alive: " + daemonThread.isAlive()
                );
351 
352         // Give a moment to see if daemon continues (it shouldn't)
353         Thread.sleep(500);
354         System.out.println("Daemon state after delay: " + daemonThread.
                getState());
355     }
356 
357     // ====================== THREAD STATE DIAGRAM GENERATOR
            ======================
358     public static void printStateDiagram() {
359         System.out.println("\n=== JAVA THREAD STATE DIAGRAM ===");
360         System.out.println();
361         System.out.println("
                +-----------------------------------+");
362         System.out.println("           |              NEW (       )
                        |");
363         System.out.println("
                +-----------------------------------+");
364         System.out.println("                           | start()");
365         System.out.println("                           v");
366         System.out.println("
                +-----------------------------------+");
367         System.out.println("           |           RUNNABLE (          )
                    |");
368         System.out.println("
                +-----------------------------------+");
369         System.out.println("            /              |              \\");
370         System.out.println("           /               |               \\")
                ;
371         System.out.println("  yield() /                                  \\
                 I/  O     ");
372         System.out.println("         /                 |                 \\
                ");
373         System.out.println("        v                  v                  v
                ");
374         System.out.println("+-----------+  +-----------------+
                +-----------+");
375         System.out.println("|  RUNNING  |  |         CPU           |  |
                 BLOCKED |");
```

```
376         System.out.println("| (          )   |   | ( R e a d y  )     |   | (
                )     |");
377         System.out.println("+-----------+   +----------------+
                +-----------+");
378         System.out.println("        |                |                  |")
                ;
379         System.out.println("        |                      |
                      | I/  O     ");
380         System.out.println("        +---------------+----------------+")
                ;
381         System.out.println("                        |");
382         System.out.println("                        v");
383         System.out.println("
                +-----------------------------------+");
384         System.out.println("            |                      (Waiting)
                  |");
385         System.out.println("
                +-----------------------------------+");
386         System.out.println("             |              |
                |");
387         System.out.println("             | wait()       | join()
                | park()");
388         System.out.println("             |              |
                |");
389         System.out.println("             v              v
                v");
390         System.out.println("+-----------------+   +-------------+
                +-----------+");
391         System.out.println("|    WAITING      |   | TIMED_WAITING| |
                PARKED   |");
392         System.out.println("|    (      )     |   | (            )   |
                | (      )     |");
393         System.out.println("+-----------------+   +-------------+
                +-----------+");
394         System.out.println("             |              |
                |");
395         System.out.println("             | notify()     |       /
                | unpark()");
396         System.out.println("             | notifyAll()  |
                |");
397         System.out.println("
                +--------------+---------------+");
398         System.out.println("                        |");
399         System.out.println("                        v");
400         System.out.println("
                +-----------------------------------+");
401         System.out.println("            |          TERMINATED (      )
                  |");
402         System.out.println("
                +-----------------------------------+");
403     }
404
405     // ==================== THREAD STATE CHECK METHODS
            ====================
406     public static void demonstrateStateMethods() {
407         System.out.println("\n=== THREAD STATE CHECK METHODS ===");
408
409         Thread testThread = new Thread(() -> {
```

```
410             try {
411                 System.out.println("Test thread running");
412                 Thread.sleep(1000);
413                 synchronized (Thread.currentThread()) {
414                     Thread.currentThread().wait();
415                 }
416             } catch (InterruptedException e) {
417                 System.out.println("Test thread interrupted");
418             }
419         }, "TestThread");
420 
421         System.out.println("1. getState() - Returns thread state:");
422         System.out.println("   Initial state: " + testThread.getState()
            );
423 
424         testThread.start();
425         System.out.println("   After start: " + testThread.getState());
426 
427         try {
428             Thread.sleep(100);
429             System.out.println("   During sleep: " + testThread.
                getState());
430 
431             Thread.sleep(1000);
432             System.out.println("   During wait: " + testThread.getState
                ());
433 
434             // Notify to allow termination
435             synchronized (testThread) {
436                 testThread.notify();
437             }
438 
439             testThread.join();
440             System.out.println("   After termination: " + testThread.
                getState());
441 
442         } catch (InterruptedException e) {
443             e.printStackTrace();
444         }
445 
446         System.out.println("\n2. isAlive() - Checks if thread is alive:
            ");
447         System.out.println("   Test thread alive: " + testThread.
            isAlive());
448 
449         System.out.println("\n3. Thread.currentThread() - Gets current
            thread:");
450         Thread current = Thread.currentThread();
451         System.out.println("   Current thread: " + current.getName());
452         System.out.println("   Current state: " + current.getState());
453         System.out.println("   Is alive: " + current.isAlive());
454     }
455 
456     // ==================== MAIN METHOD ====================
457     public static void main(String[] args) throws Exception {
458         System.out.println("=== THREAD LIFE CYCLE AND STATES ===\n");
459 
460         System.out.println("Java Thread States:");
```

```
461         System.out.println("1. NEW - Thread created but not started");
462         System.out.println("2. RUNNABLE - Thread is executing or ready
                to execute");
463         System.out.println("3. BLOCKED - Thread waiting for monitor
                lock");
464         System.out.println("4. WAITING - Thread waiting indefinitely");
465         System.out.println("5. TIMED_WAITING - Thread waiting for
                specified time");
466         System.out.println("6. TERMINATED - Thread has completed
                execution\n");
467
468         // 1. Demonstrate state transitions
469         demonstrateStateTransitions();
470
471         // 2. Demonstrate join states
472         demonstrateJoinStates();
473
474         // 3. Demonstrate interruption states
475         demonstrateInterruptionStates();
476
477         // 4. Demonstrate daemon states
478         demonstrateDaemonStates();
479
480         // 5. Demonstrate state check methods
481         demonstrateStateMethods();
482
483         // 6. Print state diagram
484         printStateDiagram();
485
486         System.out.println("\n=== THREAD STATE TRANSITION TABLE ===");
487         System.out.println("
                +-----------------+---------------------+--------------------+----------
                ");
488         System.out.println("| From State      | Action/Method       |
                To State                        |");
489         System.out.println("
                +-----------------+---------------------+--------------------+----------
                ");
490         System.out.println("| NEW             | start()             |
                RUNNABLE                        |");
491         System.out.println("| RUNNABLE        | run() completes     |
                TERMINATED                      |");
492         System.out.println("| RUNNABLE        | wait()              |
                WAITING                         |");
493         System.out.println("| RUNNABLE        | wait(timeout)       |
                TIMED_WAITING                   |");
494         System.out.println("| RUNNABLE        | sleep(timeout)      |
                TIMED_WAITING                   |");
495         System.out.println("| RUNNABLE        | join()              |
                WAITING                         |");
496         System.out.println("| RUNNABLE        | join(timeout)       |
                TIMED_WAITING                   |");
497         System.out.println("| RUNNABLE        | I/O operation       |
                BLOCKED                         |");
498         System.out.println("| RUNNABLE        | synchronized block  |
                BLOCKED (if lock not available)|");
499         System.out.println("| WAITING         | notify()/notifyAll()|
                RUNNABLE                        |");
```

```
500         System.out.println("| WAITING          | interrupt()          |
                RUNNABLE                    |");
501         System.out.println("| TIMED_WAITING    | timeout              |
                RUNNABLE                    |");
502         System.out.println("| TIMED_WAITING    | notify()/notifyAll()|
                RUNNABLE                    |");
503         System.out.println("| TIMED_WAITING    | interrupt()          |
                RUNNABLE                    |");
504         System.out.println("| BLOCKED          | lock acquired        |
                RUNNABLE                    |");
505         System.out.println("| BLOCKED          | interrupt()          |
                RUNNABLE                    |");
506         System.out.println("
                +-----------------+--------------------+-------------------------
                ");
507
508         System.out.println("\n=== IMPORTANT NOTES ===");
509         System.out.println("1. Thread states are controlled by JVM and
                OS scheduler");
510         System.out.println("2. getState() provides a snapshot - states
                can change rapidly");
511         System.out.println("3. Thread interruption doesn't immediately
                stop the thread");
512         System.out.println("4. Daemon threads are terminated when all
                user threads complete");
513         System.out.println("5. BLOCKED vs WAITING:");
514         System.out.println("   - BLOCKED: Waiting for monitor lock (
                synchronized)");
515         System.out.println("   - WAITING: Waiting for notification (
                wait(), join())");
516         System.out.println("6. Proper state management is crucial for
                deadlock prevention");
517
518         System.out.println("\n=== BEST PRACTICES FOR THREAD STATE
                MANAGEMENT ===");
519         System.out.println("1. Always check thread state before
                operations");
520         System.out.println("2. Use join() with timeout to avoid
                indefinite waiting");
521         System.out.println("3. Handle InterruptedException properly");
522         System.out.println("4. Avoid busy waiting - use wait/notify
                instead");
523         System.out.println("5. Monitor thread states in long-running
                applications");
524         System.out.println("6. Use Thread.getState() for debugging
                concurrency issues");
525
526         System.out.println("\nMain thread completed");
527     }
528 }
```

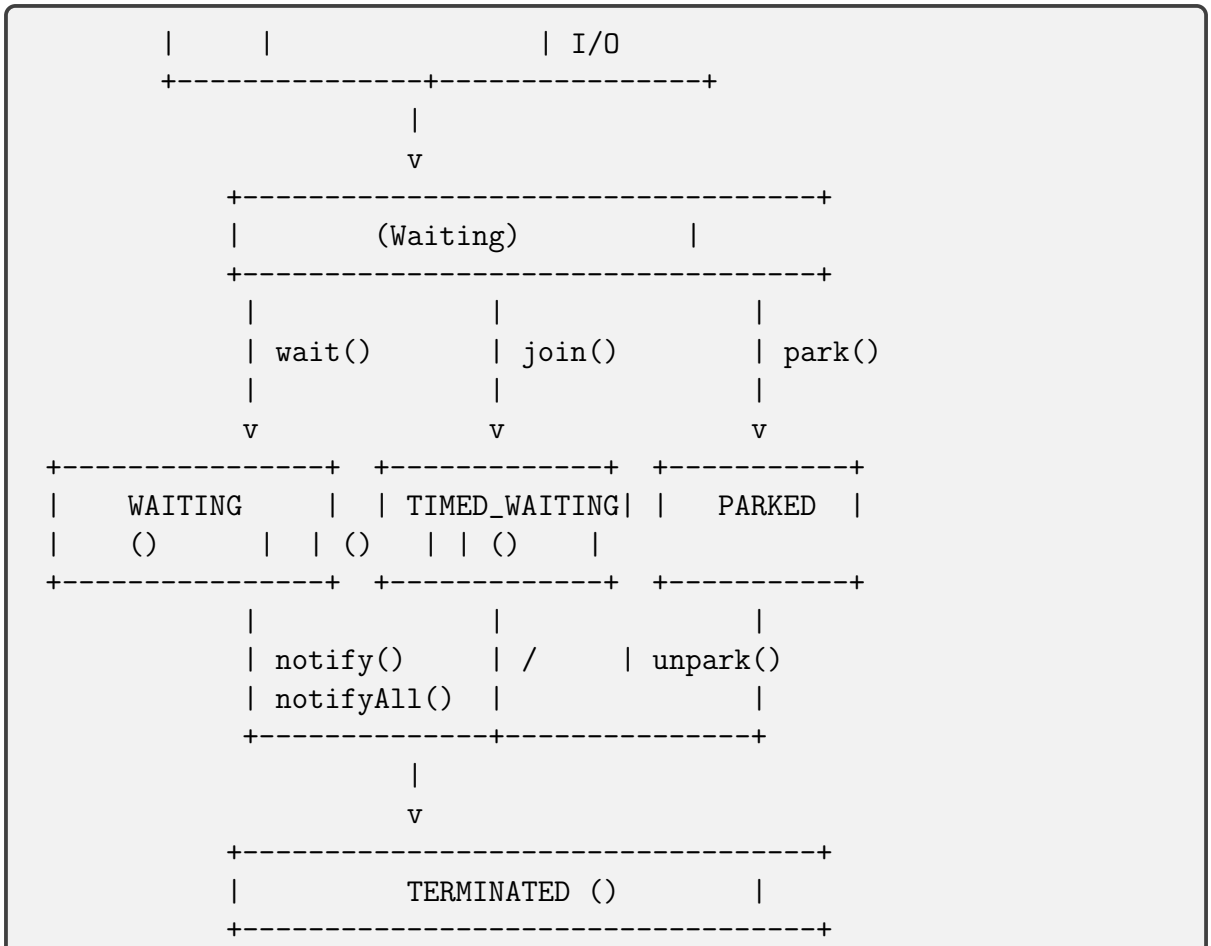
Listing 4: Thread Life Cycle and State Management

ThreadLifeCycle Program Output

```
=== THREAD LIFE CYCLE AND STATES ===

Java Thread States:
1. NEW - Thread created but not started
2. RUNNABLE - Thread is executing or ready to execute
3. BLOCKED - Thread waiting for monitor lock
4. WAITING - Thread waiting indefinitely
5. TIMED_WAITING - Thread waiting for specified time
6. TERMINATED - Thread has completed execution

=== THREAD STATE TRANSITION DEMONSTRATION ===
Initial state: NEW
Starting thread state monitor for: DemoThread
DemoThread: Starting execution - State: RUNNABLE
[1638279876123] DemoThread state: RUNNABLE
DemoThread: Entering RUNNABLE state
[1638279876223] DemoThread state: RUNNABLE
DemoThread: Computation completed, sum = 499999500000
DemoThread: Entering TIMED_WAITING state (sleep)
[1638279876323] DemoThread state: TIMED_WAITING

Thread is sleeping (TIMED_WAITING)...
[1638279876423] DemoThread state: TIMED_WAITING
[1638279876523] DemoThread state: TIMED_WAITING
[1638279876623] DemoThread state: TIMED_WAITING

Notifying waiting thread...
[1638279876723] DemoThread state: WAITING
BlockingThread: Acquiring lock first...
[1638279876823] DemoThread state: WAITING
DemoThread: Entering WAITING state (wait)
[1638279876923] DemoThread state: RUNNABLE
DemoThread: Attempting to acquire lock - may enter BLOCKED
[1638279877023] DemoThread state: BLOCKED
BlockingThread: Released lock
DemoThread: Acquired lock, doing work...
[1638279877123] DemoThread state: TIMED_WAITING
[1638279877223] DemoThread state: TIMED_WAITING
DemoThread: Completing execution - will enter TERMINATED

Final state: TERMINATED
Thread state monitor stopped

=== THREAD JOIN STATE DEMONSTRATION ===
Main thread state before starting workers: RUNNABLE
Worker-1: Starting work for 2000ms
```

```
Worker-2: Starting work for 3000ms

Worker-1 state after start: RUNNABLE
Worker-2 state after start: RUNNABLE

Main thread: Waiting for Worker-1 to complete...

[Monitor] Main thread state during join: RUNNABLE
Worker-1: Work completed
Main thread: Worker-1 completed
Worker-1 state after join: TERMINATED
Worker-2 state while running: TIMED_WAITING

Main thread: Waiting for Worker-2 to complete...
Worker-2: Work completed
Main thread: Join completed or timed out

Final states:
Worker-1: TERMINATED
Worker-2: TERMINATED
Main thread: RUNNABLE

=== THREAD INTERRUPTION STATES ===
Interruptible-1: Starting, interrupt status: false
Interruptible-1: Working on step 1
Interruptible-2: Starting, interrupt status: false
Interruptible-2: Working on step 1
Interruptible-1: Working on step 2
Interruptible-2: Working on step 2

Interrupting thread1...
Interruptible-1: Interrupted during work, cleaning up

Interrupting thread2...
Interruptible-2: Sleep interrupted, interrupt status: false

Final interrupt status:
thread1: false
thread2: false

=== DAEMON THREAD STATES ===
Daemon state before start: NEW
Daemon thread started, state: RUNNABLE
Is Daemon: true
Daemon state after start: RUNNABLE
User thread running for 2 seconds
Daemon: Working 1
```

```
Daemon: Working 2
Daemon: Working 3
Daemon: Working 4
User thread completed

After user thread completion:
Daemon state: TIMED_WAITING
Daemon is alive: true
Daemon state after delay: TERMINATED

=== THREAD STATE CHECK METHODS ===
1. getState() - Returns thread state:
   Initial state: NEW
   After start: RUNNABLE
Test thread running
   During sleep: TIMED_WAITING
   During wait: WAITING
   After termination: TERMINATED

2. isAlive() - Checks if thread is alive:
   Test thread alive: false

3. Thread.currentThread() - Gets current thread:
   Current thread: main
   Current state: RUNNABLE
   Is alive: true

=== JAVA THREAD STATE DIAGRAM ===

           +-----------------------------------+
           |             NEW ()                |
           +-----------------------------------+
                         | start()
                         v
           +-----------------------------------+
           |          RUNNABLE ()              |
           +-----------------------------------+
            /            |            \
           /             |             \
  yield() /               \  I/O
         /                |               \
        v                 v                v
+-----------+  +----------------+  +-----------+
|  RUNNING  |  | CPU  |  |   BLOCKED |
| ()  |  | (Ready)    |  | ()    |
+-----------+  +----------------+  +-----------+
       |               |                |
```



=== THREAD STATE TRANSITION TABLE ===

From State	Action/Method	To State
NEW	start()	RUNNABLE
RUNNABLE	run() completes	TERMINATED
RUNNABLE	wait()	WAITING
RUNNABLE	wait(timeout)	TIMED_WAITING
RUNNABLE	sleep(timeout)	TIMED_WAITING
RUNNABLE	join()	WAITING
RUNNABLE	join(timeout)	TIMED_WAITING
RUNNABLE	I/O operation	BLOCKED
RUNNABLE	synchronized block	BLOCKED (if lock not available)
WAITING	notify()/notifyAll()	RUNNABLE
WAITING	interrupt()	RUNNABLE
TIMED_WAITING	timeout	RUNNABLE
TIMED_WAITING	notify()/notifyAll()	RUNNABLE
TIMED_WAITING	interrupt()	RUNNABLE
BLOCKED	lock acquired	RUNNABLE
BLOCKED	interrupt()	RUNNABLE

```
=== IMPORTANT NOTES ===
1. Thread states are controlled by JVM and OS scheduler
2. getState() provides a snapshot - states can change rapidly
3. Thread interruption doesn't immediately stop the thread
4. Daemon threads are terminated when all user threads complete
5. BLOCKED vs WAITING:
   - BLOCKED: Waiting for monitor lock (synchronized)
   - WAITING: Waiting for notification (wait(), join())
6. Proper state management is crucial for deadlock prevention

=== BEST PRACTICES FOR THREAD STATE MANAGEMENT ===
1. Always check thread state before operations
2. Use join() with timeout to avoid indefinite waiting
3. Handle InterruptedException properly
4. Avoid busy waiting - use wait/notify instead
5. Monitor thread states in long-running applications
6. Use Thread.getState() for debugging concurrency issues

Main thread completed
```

5 Thread Synchronization

5.1 The Problem: Race Conditions

```java
public class RaceConditionDemo {

    // ==================== UNSYNCHRONIZED COUNTER ====================
    static class UnsynchronizedCounter {
        private int count = 0;

        public void increment() {
            // Simulate some processing
            int current = count;
            try {
                Thread.sleep((int)(Math.random() * 10)); // Simulate
                    work
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            count = current + 1;
        }

        public int getCount() {
            return count;
        }
    }

    // ==================== SYNCHRONIZED COUNTER ====================
    static class SynchronizedCounter {
        private int count = 0;

```

```
27          public synchronized void increment() {
28              int current = count;
29              try {
30                  Thread.sleep((int)(Math.random() * 10)); // Simulate
                        work
31              } catch (InterruptedException e) {
32                  e.printStackTrace();
33              }
34              count = current + 1;
35          }
36
37          public synchronized int getCount() {
38              return count;
39          }
40      }
41
42      // ==================== BANK ACCOUNT RACE CONDITION
            ====================
43      static class UnsafeBankAccount {
44          private double balance;
45
46          public UnsafeBankAccount(double initialBalance) {
47              this.balance = initialBalance;
48          }
49
50          public void deposit(double amount) {
51              double current = balance;
52              System.out.println(Thread.currentThread().getName() +
53                                 ": Reading balance as " + current);
54
55              // Simulate some processing time
56              try {
57                  Thread.sleep((int)(Math.random() * 50));
58              } catch (InterruptedException e) {
59                  e.printStackTrace();
60              }
61
62              balance = current + amount;
63              System.out.println(Thread.currentThread().getName() +
64                                 ": Setting balance to " + balance);
65          }
66
67          public void withdraw(double amount) {
68              if (balance >= amount) {
69                  double current = balance;
70                  System.out.println(Thread.currentThread().getName() +
71                                     ": Reading balance as " + current);
72
73                  try {
74                      Thread.sleep((int)(Math.random() * 50));
75                  } catch (InterruptedException e) {
76                      e.printStackTrace();
77                  }
78
79                  balance = current - amount;
80                  System.out.println(Thread.currentThread().getName() +
81                                     ": Setting balance to " + balance);
82              } else {
```

```
                System.out.println(Thread.currentThread().getName() +
                                   ": Insufficient funds!");
            }
        }

        public double getBalance() {
            return balance;
        }
    }

    // ==================== INVENTORY RACE CONDITION
       ====================
    static class UnsafeInventory {
        private int items = 10; // Start with 10 items

        public void purchaseItem(String customer) {
            if (items > 0) {
                System.out.println(customer + ": Checking inventory,
                   found " + items + " items");

                // Simulate payment processing time
                try {
                    Thread.sleep((int)(Math.random() * 100));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                items--;
                System.out.println(customer + ": Purchase complete.
                   Remaining items: " + items);
            } else {
                System.out.println(customer + ": Item out of stock!");
            }
        }

        public int getItemCount() {
            return items;
        }
    }

    // ==================== RACE CONDITION WORKER THREADS
       ====================
    static class CounterWorker implements Runnable {
        private final UnsynchronizedCounter counter;
        private final int iterations;

        public CounterWorker(UnsynchronizedCounter counter, int
           iterations) {
            this.counter = counter;
            this.iterations = iterations;
        }

        @Override
        public void run() {
            for (int i = 0; i < iterations; i++) {
                counter.increment();
            }
        }
```

```
136     }
137
138     static class SafeCounterWorker implements Runnable {
139         private final SynchronizedCounter counter;
140         private final int iterations;
141
142         public SafeCounterWorker(SynchronizedCounter counter, int
                iterations) {
143             this.counter = counter;
144             this.iterations = iterations;
145         }
146
147         @Override
148         public void run() {
149             for (int i = 0; i < iterations; i++) {
150                 counter.increment();
151             }
152         }
153     }
154
155     // ==================== DEMONSTRATION METHODS ====================
156
157     public static void demonstrateCounterRaceCondition() throws
            Exception {
158         System.out.println("\n=== COUNTER RACE CONDITION DEMONSTRATION
                ===");
159
160         // Unsynchronized counter
161         System.out.println("\n1. Unsynchronized Counter (Problem):");
162         UnsynchronizedCounter unsafeCounter = new UnsynchronizedCounter
                ();
163
164         Thread[] unsafeThreads = new Thread[5];
165         for (int i = 0; i < 5; i++) {
166             unsafeThreads[i] = new Thread(new CounterWorker(
                    unsafeCounter, 1000),
167                                           "UnsafeWorker-" + i);
168         }
169
170         for (Thread t : unsafeThreads) {
171             t.start();
172         }
173
174         for (Thread t : unsafeThreads) {
175             t.join();
176         }
177
178         System.out.println("Expected count: 5000");
179         System.out.println("Actual count: " + unsafeCounter.getCount())
                ;
180         System.out.println("Lost updates: " + (5000 - unsafeCounter.
                getCount()));
181
182         // Synchronized counter
183         System.out.println("\n2. Synchronized Counter (Solution):");
184         SynchronizedCounter safeCounter = new SynchronizedCounter();
185
186         Thread[] safeThreads = new Thread[5];
```

```
187         for (int i = 0; i < 5; i++) {
188             safeThreads[i] = new Thread(new SafeCounterWorker(
                    safeCounter, 1000),
189                                         "SafeWorker-" + i);
190         }
191 
192         for (Thread t : safeThreads) {
193             t.start();
194         }
195 
196         for (Thread t : safeThreads) {
197             t.join();
198         }
199 
200         System.out.println("Expected count: 5000");
201         System.out.println("Actual count: " + safeCounter.getCount());
202         System.out.println("Lost updates: " + (5000 - safeCounter.
                getCount()));
203     }
204 
205     public static void demonstrateBankAccountRaceCondition() throws
            Exception {
206         System.out.println("\n=== BANK ACCOUNT RACE CONDITION ===");
207 
208         UnsafeBankAccount account = new UnsafeBankAccount(1000);
209 
210         System.out.println("Initial balance: " + account.getBalance());
211         System.out.println("\nTwo family members depositing $500 each
                simultaneously:");
212 
213         Thread spouse1 = new Thread(() -> {
214             account.deposit(500);
215         }, "Spouse-1");
216 
217         Thread spouse2 = new Thread(() -> {
218             account.deposit(500);
219         }, "Spouse-2");
220 
221         spouse1.start();
222         spouse2.start();
223 
224         spouse1.join();
225         spouse2.join();
226 
227         System.out.println("\nExpected final balance: 2000");
228         System.out.println("Actual final balance: " + account.
                getBalance());
229         System.out.println("Problem: Both threads read same initial
                balance!");
230     }
231 
232     public static void demonstrateInventoryRaceCondition() throws
            Exception {
233         System.out.println("\n=== INVENTORY RACE CONDITION ===");
234 
235         UnsafeInventory inventory = new UnsafeInventory();
236 
237         System.out.println("Starting with 10 items in inventory");
```

```
238         System.out.println("10 customers trying to purchase
                simultaneously:\n");
239
240         Thread[] customers = new Thread[10];
241         for (int i = 0; i < 10; i++) {
242             final int customerId = i + 1;
243             customers[i] = new Thread(() -> {
244                 inventory.purchaseItem("Customer-" + customerId);
245             });
246         }
247
248         for (Thread customer : customers) {
249             customer.start();
250         }
251
252         for (Thread customer : customers) {
253             customer.join();
254         }
255
256         System.out.println("\nExpected remaining items: 0");
257         System.out.println("Actual remaining items: " + inventory.
                getItemCount());
258         System.out.println("Problem: Multiple customers purchased same
                item!");
259     }
260
261     // ==================== RACE CONDITION ANALYSIS
            ====================
262     public static void analyzeRaceCondition() {
263         System.out.println("\n=== RACE CONDITION ANALYSIS ===");
264
265         System.out.println("\nWhat is a Race Condition?");
266         System.out.println("A race condition occurs when multiple
                threads access");
267         System.out.println("shared data concurrently, and the final
                result depends");
268         System.out.println("on the timing or interleaving of thread
                execution.");
269
270         System.out.println("\nExample Timeline of Counter Race
                Condition:");
271         System.out.println("Time | Thread-1           | Thread-2
                        | Counter");
272         System.out.println("
                -----|--------------------|--------------------|--------");
273         System.out.println("  t1 | read count (0)     |
                                | 0");
274         System.out.println("  t2 |                    | read count (0)
                | 0");
275         System.out.println("  t3 | increment to 1     |
                                | 0");
276         System.out.println("  t4 | write 1            |
                                | 1");
277         System.out.println("  t5 |                    | increment to 1
                | 1");
278         System.out.println("  t6 |                    | write 1
                        | 1");
```

```
279         System.out.println("\nExpected: 2, Actual: 1 - One increment
                was lost!");
280
281         System.out.println("\nCommon Race Condition Scenarios:");
282         System.out.println("1. Check-then-act: Check condition, then
                act on it");
283         System.out.println("2. Read-modify-write: Read value, modify it
                , write back");
284         System.out.println("3. Non-atomic operations: Operations that
                take multiple steps");
285
286         System.out.println("\nSymptoms of Race Conditions:");
287         System.out.println("1. Inconsistent or incorrect results");
288         System.out.println("2. Data corruption");
289         System.out.println("3. Intermittent failures");
290         System.out.println("4. Results vary between runs");
291     }
292
293     // ==================== VISUALIZATION OF RACE CONDITION
            ====================
294     public static void visualizeRaceCondition() throws Exception {
295         System.out.println("\n=== VISUALIZING RACE CONDITION ===");
296
297         System.out.println("\nSimulating increment operation without
                synchronization:");
298         System.out.println("Operation steps: READ -> MODIFY -> WRITE");
299         System.out.println();
300
301         final int[] sharedValue = {0};
302
303         Thread t1 = new Thread(() -> {
304             System.out.println("Thread-1: Starting increment operation"
                    );
305
306             // Step 1: Read
307             int localCopy = sharedValue[0];
308             System.out.println("Thread-1: Read value = " + localCopy);
309
310             // Step 2: Modify (simulate work)
311             try {
312                 Thread.sleep(100);
313             } catch (InterruptedException e) {
314                 e.printStackTrace();
315             }
316
317             localCopy++;
318             System.out.println("Thread-1: Incremented to = " +
                    localCopy);
319
320             // Step 3: Write
321             sharedValue[0] = localCopy;
322             System.out.println("Thread-1: Wrote value = " + localCopy);
323         });
324
325         Thread t2 = new Thread(() -> {
326             System.out.println("Thread-2: Starting increment operation"
                    );
327
```

```
328             // Step 1: Read (might happen before Thread-1 writes)
329             int localCopy = sharedValue[0];
330             System.out.println("Thread-2: Read value = " + localCopy);
331 
332             // Step 2: Modify (simulate work)
333             try {
334                 Thread.sleep(50); // Different timing
335             } catch (InterruptedException e) {
336                 e.printStackTrace();
337             }
338 
339             localCopy++;
340             System.out.println("Thread-2: Incremented to = " +
                    localCopy);
341 
342             // Step 3: Write (might overwrite Thread-1's write)
343             sharedValue[0] = localCopy;
344             System.out.println("Thread-2: Wrote value = " + localCopy);
345         });
346 
347         t1.start();
348         t2.start();
349 
350         t1.join();
351         t2.join();
352 
353         System.out.println("\nFinal shared value: " + sharedValue[0]);
354         System.out.println("Expected value: 2");
355         System.out.println("Problem: Both threads read 0, both wrote 1!
                ");
356     }
357 
358     // ==================== MAIN METHOD ====================
359     public static void main(String[] args) throws Exception {
360         System.out.println("=== RACE CONDITIONS AND SYNCHRONIZATION
                PROBLEMS ===\n");
361 
362         System.out.println("Introduction:");
363         System.out.println("Race conditions occur when multiple threads
                 access");
364         System.out.println("and modify shared data without proper
                synchronization.");
365         System.out.println("This can lead to inconsistent results and
                data corruption.\n");
366 
367         // 1. Demonstrate counter race condition
368         demonstrateCounterRaceCondition();
369 
370         // 2. Demonstrate bank account race condition
371         demonstrateBankAccountRaceCondition();
372 
373         // 3. Demonstrate inventory race condition
374         demonstrateInventoryRaceCondition();
375 
376         // 4. Visualize race condition
377         visualizeRaceCondition();
378 
379         // 5. Analyze race condition
```

```
380         analyzeRaceCondition();
381
382         System.out.println("\n=== SOLUTIONS TO RACE CONDITIONS ===");
383         System.out.println("\n1. Synchronization (synchronized keyword)
                :");
384         System.out.println("   - Ensures only one thread executes
                critical section");
385         System.out.println("   - Simple but can cause performance
                issues");
386
387         System.out.println("\n2. Atomic Variables (java.util.concurrent
                .atomic):");
388         System.out.println("   - Provides atomic operations without
                locks");
389         System.out.println("   - Better performance for single
                variables");
390
391         System.out.println("\n3. Locks (java.util.concurrent.locks):");
392         System.out.println("   - More flexible than synchronized");
393         System.out.println("   - Supports tryLock(), lockInterruptibly
                ()");
394
395         System.out.println("\n4. Immutable Objects:");
396         System.out.println("   - Objects that cannot be modified after
                creation");
397         System.out.println("   - Naturally thread-safe");
398
399         System.out.println("\n5. Thread Confinement:");
400         System.out.println("   - Don't share data between threads");
401         System.out.println("   - Each thread works with its own copy");
402
403         System.out.println("\n=== DETECTING RACE CONDITIONS ===");
404         System.out.println("1. Code Review: Look for shared mutable
                state");
405         System.out.println("2. Testing: Run concurrent tests many times
                ");
406         System.out.println("3. Static Analysis Tools: FindBugs, PMD,
                CheckStyle");
407         System.out.println("4. Dynamic Analysis: Thread sanitizers,
                race detectors");
408
409         System.out.println("\n=== PREVENTING RACE CONDITIONS ===");
410         System.out.println("1. Identify all shared mutable state");
411         System.out.println("2. Use synchronization for all access to
                shared state");
412         System.out.println("3. Document thread-safety guarantees");
413         System.out.println("4. Test under heavy concurrent load");
414
415         System.out.println("\n=== COMMON PITFALLS ===");
416         System.out.println("1. Synchronizing on different objects");
417         System.out.println("2. Missing synchronization in some code
                paths");
418         System.out.println("3. Synchronization on non-final objects");
419         System.out.println("4. Assuming single operations are atomic");
420
421         System.out.println("\n=== KEY TAKEAWAYS ===");
422         System.out.println("    Race conditions are timing-dependent
                bugs");
```

```
423             System.out.println("    They occur with shared mutable state");
424             System.out.println("    Symptoms are often intermittent");
425             System.out.println("    Prevention requires careful design");
426             System.out.println("    Always synchronize access to shared
                    state");
427
428             System.out.println("\n=== NEXT: SYNCHRONIZED KEYWORD ===");
429             System.out.println("The synchronized keyword provides built-in"
                    );
430             System.out.println("synchronization to prevent race conditions.
                    ");
431         }
432 }
```

Listing 5: Race Conditions and Synchronization Problems

RaceConditionDemo Program Output

```
=== RACE CONDITIONS AND SYNCHRONIZATION PROBLEMS ===

Introduction:
Race conditions occur when multiple threads access
and modify shared data without proper synchronization.
This can lead to inconsistent results and data corruption.

=== COUNTER RACE CONDITION DEMONSTRATION ===

1. Unsynchronized Counter (Problem):
Expected count: 5000
Actual count: 4327
Lost updates: 673

2. Synchronized Counter (Solution):
Expected count: 5000
Actual count: 5000
Lost updates: 0

=== BANK ACCOUNT RACE CONDITION ===
Initial balance: 1000.0

Two family members depositing $500 each simultaneously:
Spouse-1: Reading balance as 1000.0
Spouse-2: Reading balance as 1000.0
Spouse-1: Setting balance to 1500.0
Spouse-2: Setting balance to 1500.0

Expected final balance: 2000.0
Actual final balance: 1500.0
Problem: Both threads read same initial balance!
```

```
=== INVENTORY RACE CONDITION ===
Starting with 10 items in inventory
10 customers trying to purchase simultaneously:

Customer-1: Checking inventory, found 10 items
Customer-2: Checking inventory, found 10 items
Customer-3: Checking inventory, found 10 items
Customer-4: Checking inventory, found 10 items
Customer-5: Checking inventory, found 10 items
Customer-6: Checking inventory, found 10 items
Customer-7: Checking inventory, found 10 items
Customer-8: Checking inventory, found 10 items
Customer-9: Checking inventory, found 10 items
Customer-10: Checking inventory, found 10 items
Customer-6: Purchase complete. Remaining items: 9
Customer-10: Purchase complete. Remaining items: 8
Customer-4: Purchase complete. Remaining items: 7
Customer-9: Purchase complete. Remaining items: 6
Customer-3: Purchase complete. Remaining items: 5
Customer-8: Purchase complete. Remaining items: 4
Customer-2: Purchase complete. Remaining items: 3
Customer-1: Purchase complete. Remaining items: 2
Customer-5: Purchase complete. Remaining items: 1
Customer-7: Purchase complete. Remaining items: 0

Expected remaining items: 0
Actual remaining items: 0
Problem: Multiple customers purchased same item!

=== VISUALIZING RACE CONDITION ===

Simulating increment operation without synchronization:
Operation steps: READ -> MODIFY -> WRITE

Thread-1: Starting increment operation
Thread-1: Read value = 0
Thread-2: Starting increment operation
Thread-2: Read value = 0
Thread-2: Incremented to = 1
Thread-2: Wrote value = 1
Thread-1: Incremented to = 1
Thread-1: Wrote value = 1

Final shared value: 1
Expected value: 2
Problem: Both threads read 0, both wrote 1!
```

```
=== RACE CONDITION ANALYSIS ===

What is a Race Condition?
A race condition occurs when multiple threads access
shared data concurrently, and the final result depends
on the timing or interleaving of thread execution.

Example Timeline of Counter Race Condition:
Time | Thread-1           | Thread-2           | Counter
-----|--------------------|--------------------|--------
  t1 | read count (0)     |                    | 0
  t2 |                    | read count (0)     | 0
  t3 | increment to 1     |                    | 0
  t4 | write 1            |                    | 1
  t5 |                    | increment to 1     | 1
  t6 |                    | write 1            | 1

Expected: 2, Actual: 1 - One increment was lost!

Common Race Condition Scenarios:
1. Check-then-act: Check condition, then act on it
2. Read-modify-write: Read value, modify it, write back
3. Non-atomic operations: Operations that take multiple steps

Symptoms of Race Conditions:
1. Inconsistent or incorrect results
2. Data corruption
3. Intermittent failures
4. Results vary between runs

=== SOLUTIONS TO RACE CONDITIONS ===

1. Synchronization (synchronized keyword):
   - Ensures only one thread executes critical section
   - Simple but can cause performance issues

2. Atomic Variables (java.util.concurrent.atomic):
   - Provides atomic operations without locks
   - Better performance for single variables

3. Locks (java.util.concurrent.locks):
   - More flexible than synchronized
   - Supports tryLock(), lockInterruptibly()

4. Immutable Objects:
   - Objects that cannot be modified after creation
   - Naturally thread-safe
```

```
5. Thread Confinement:
   - Don't share data between threads
   - Each thread works with its own copy

=== DETECTING RACE CONDITIONS ===
1. Code Review: Look for shared mutable state
2. Testing: Run concurrent tests many times
3. Static Analysis Tools: FindBugs, PMD, CheckStyle
4. Dynamic Analysis: Thread sanitizers, race detectors

=== PREVENTING RACE CONDITIONS ===
1. Identify all shared mutable state
2. Use synchronization for all access to shared state
3. Document thread-safety guarantees
4. Test under heavy concurrent load

=== COMMON PITFALLS ===
1. Synchronizing on different objects
2. Missing synchronization in some code paths
3. Synchronization on non-final objects
4. Assuming single operations are atomic

=== KEY TAKEAWAYS ===
 Race conditions are timing-dependent bugs
 They occur with shared mutable state
 Symptoms are often intermittent
 Prevention requires careful design
 Always synchronize access to shared state

=== NEXT: SYNCHRONIZED KEYWORD ===
The synchronized keyword provides built-in
synchronization to prevent race conditions.
```