

UNIT 7

Threads

(3 Hours) — Object Oriented Programming in Java (CAC204)

7.1 Introduction to Threads

A **thread** is the smallest unit of execution inside a program. A **process** is a running program with its own memory space; a single process can contain many threads that share the same memory but each run their own path of instructions. Every Java program has at least one thread — the **main thread** — which starts running `main()` when the JVM launches.

Multithreading means executing two or more threads concurrently within one program. Examples: a word processor that checks spelling while you type, a browser downloading a file while rendering a page, or a game running graphics, sound and input handling together.

Process vs Thread

Basis	Process	Thread
Definition	A program in execution	A lightweight sub-unit of a process
Memory	Has its own separate address space	Shares the memory (heap) of its process
Creation cost	Heavy — more resources and time	Light — less resources and time
Communication	Difficult (needs inter-process communication)	Easy (shared objects and variables)
Context switch	Slow	Fast
Failure effect	Independent of other processes	An unhandled problem may affect the whole process

Advantages of Multithreading

- **Better resource use** — the CPU is not idle while one thread waits for input/output.
- **Responsiveness** — a long task in the background does not freeze the user interface.
- **Faster execution** — independent tasks can run in parallel on multi-core processors.
- **Economical** — threads share memory, so they are cheaper than processes.

*Note: On a single-core CPU threads are not truly simultaneous; the CPU switches rapidly between them (time slicing), creating the illusion of parallelism. This is called **concurrency**. On multi-core CPUs threads can run in real **parallel**.*

7.2 Creating and Starting New Threads

Java supports threads through the class `java.lang.Thread` and the interface `java.lang.Runnable`. There are two classic ways to create a thread; in both, the code that the thread should execute is placed in the `run()` method.

Method 1: Extending the `java.lang.Thread` Class

Create a subclass of `Thread`, override `run()`, create an object of the subclass and call `start()`.

Program 7.1 — Extending Thread

```
class MyThread extends Thread {
    public void run() {                // code executed by the new thread
        for (int i = 1; i <= 3; i++) {
            System.out.println(getName() + " prints " + i);
        }
    }
}

public class ThreadDemo1 {
    public static void main(String[] args) {
        MyThread t1 = new MyThread(); // state: NEW
        MyThread t2 = new MyThread();
        t1.start();                    // state: RUNNABLE
        t2.start();
    }
}
```

Sample Output (order may vary on every run)

```
Thread-0 prints 1 Thread-1 prints 1 Thread-0 prints 2 Thread-1 prints 2
...
```

The output interleaves because the two threads run concurrently and the scheduler decides who runs when.

Method 2: Implementing the java.lang.Runnable Interface

Create a class that implements `Runnable` and defines `run()`. Then pass an object of that class to a `Thread` constructor and call `start()` on the `Thread` object.

Program 7.2 — Implementing Runnable

```
class Task implements Runnable {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(Thread.currentThread().getName() + " : " + i);
        }
    }
}

public class ThreadDemo2 {
    public static void main(String[] args) {
        Task job = new Task();
        Thread t1 = new Thread(job, "Worker-A"); // name is optional
        Thread t2 = new Thread(job, "Worker-B");
        t1.start();
        t2.start();

        // Runnable is a functional interface, so a lambda also works:
        new Thread(() -> System.out.println("Hello from lambda")).start();
    }
}
```

Extending Thread vs Implementing Runnable

Basis	extends Thread	implements Runnable
Inheritance	Uses up the single superclass slot (Java has no multiple inheritance)	Class is free to extend another class
Design	Thread and the task are tightly coupled	Separates the task (what) from the runner (how)
Object sharing	Each thread is a separate object	One Runnable object can be shared by many threads
Flexibility	Less flexible	More flexible; works with thread pools and lambdas
Preferred?	Simple demos only	Yes — recommended in practice

start() versus run()

- **start()** creates a new call stack, moves the thread to the runnable state, and lets the JVM call `run()` in the *new* thread.
- Calling **run()** directly is just an ordinary method call — it executes in the *current* thread and no new thread is created.
- Calling `start()` twice on the same thread throws `IllegalThreadStateException`.

Commonly Used Thread Methods

Method	Purpose
<code>void start()</code>	Begins execution; JVM calls <code>run()</code>
<code>void run()</code>	Entry point containing the thread's task
<code>static void sleep(long ms)</code>	Pauses the current thread for the given milliseconds
<code>void join()</code>	Makes the caller wait until this thread finishes
<code>static void yield()</code>	Hints that the current thread is willing to give up the CPU
<code>boolean isAlive()</code>	true if the thread has started and not yet terminated
<code>String getName() / setName()</code>	Gets or sets the thread's name
<code>int getPriority() / setPriority()</code>	Gets or sets the priority (1 to 10)
<code>static Thread currentThread()</code>	Returns a reference to the currently executing thread
<code>void interrupt()</code>	Requests the thread to stop what it is doing
<code>void setDaemon(boolean)</code>	Marks the thread as a daemon (background) thread

7.3 Understanding Thread Execution (Life Cycle)

During its life a thread passes through several states, defined by the enum `Thread.State`. The state at any moment can be read with `getState()`.

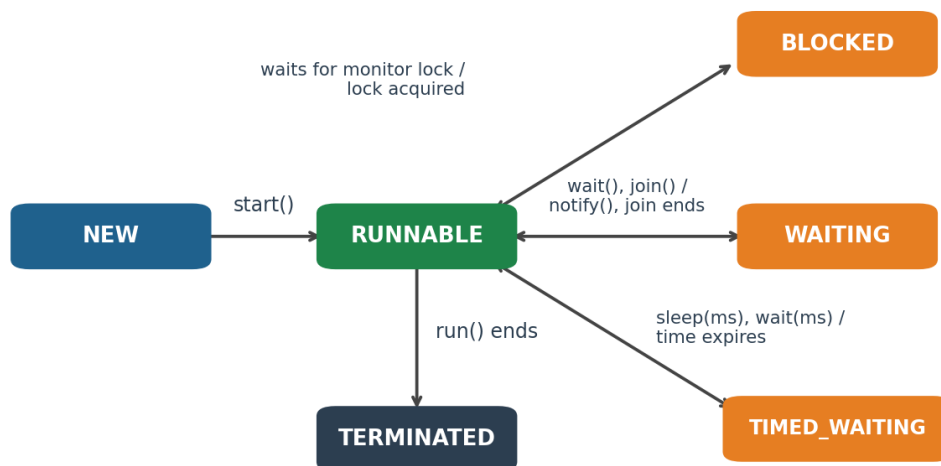


Figure 7.1 — Life cycle (states) of a Java thread

State	Meaning
NEW	Thread object created with <code>new</code> , but <code>start()</code> not yet called.
RUNNABLE	Ready to run or currently running on the CPU; waits for the scheduler to give it time.
BLOCKED	Waiting to acquire a monitor lock held by another thread (e.g. to enter a synchronized block).
WAITING	Waiting indefinitely for another thread — after <code>wait()</code> or <code>join()</code> with no timeout.
TIMED_WAITING	Waiting for a fixed time — after <code>sleep(ms)</code> , <code>wait(ms)</code> or <code>join(ms)</code> .
TERMINATED	The <code>run()</code> method has completed (normally or by an uncaught exception). It cannot be restarted.

Program 7.3 — sleep() and join()

```
class Counter extends Thread {
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(getName() + " : " + i);
            try {
                Thread.sleep(500);           // TIMED_WAITING for 0.5 s
            } catch (InterruptedException e) {
                System.out.println("Interrupted");
            }
        }
    }
}

public class JoinDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();
        System.out.println("State: " + c.getState());    // NEW
        c.start();
        c.join();                                         // main waits until c terminates
        System.out.println("State: " + c.getState());    // TERMINATED
        System.out.println("main ends after Counter");
    }
}
```

Note: Both `sleep()` and `join()` throw the checked exception `InterruptedException`, so they must be enclosed in `try-catch` or the method must declare `throws`.

How the Scheduler Decides

Thread scheduling is handled by the JVM together with the operating system. Most systems use **preemptive scheduling with time slicing**: a running thread is interrupted when its time slice ends or a higher-priority thread becomes ready. Because the exact order is not defined by the Java language, a program must never depend on the order in which threads run.

7.4 Thread Priorities

Every thread has a **priority**, an integer from 1 to 10, which is a hint to the scheduler about how important the thread is. A thread inherits the priority of the thread that created it (normally 5).

Constant	Value	Meaning
<code>Thread.MIN_PRIORITY</code>	1	Lowest priority
<code>Thread.NORM_PRIORITY</code>	5	Default priority
<code>Thread.MAX_PRIORITY</code>	10	Highest priority

Program 7.4 — Setting priorities

```
public class PriorityDemo {
    public static void main(String[] args) {
        Thread low  = new Thread(() -> System.out.println("Low priority"));
        Thread high = new Thread(() -> System.out.println("High priority"));
        low.setPriority(Thread.MIN_PRIORITY);      // 1
        high.setPriority(Thread.MAX_PRIORITY);     // 10
        System.out.println(low.getPriority() + " " + high.getPriority());
        low.start();
        high.start();      // usually (not always) gets the CPU earlier
    }
}
```

*Note: Priority is only a **hint**. Higher priority does not guarantee earlier or more frequent execution; behaviour differs between operating systems. `setPriority()` throws `IllegalArgumentException` if the value is outside 1–10.*

7.5 Synchronization

Threads share the same heap memory. When two or more threads read and modify the same data at the same time, the result may be wrong and unpredictable. This problem is called a **race condition**.

Example — Race condition on a shared counter

Two threads each execute `count++` 1000 times on the same variable. The expected final value is 2000.

But `count++` is really three steps: **read** → **add 1** → **write back**. If both threads read the same old value before either writes, one update is lost. The final value may be 1874, 1960, etc.

The synchronized Keyword

Synchronization allows only **one thread at a time** to execute a critical section of code. Every Java object has an internal **monitor (intrinsic lock)**. A thread must own the object's lock to enter a synchronized method or block; other threads that want the same lock enter the **BLOCKED** state until it is released. This property is called **mutual exclusion**.

Program 7.5 — Fixing the race condition

```
class Counter {
    private int count = 0;
    public synchronized void increment() {        // synchronized method
        count++;
    }
    public int getCount() { return count; }
}

public class SyncDemo {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();
        Runnable job = () -> {
            for (int i = 0; i < 1000; i++) c.increment();
        };
        Thread t1 = new Thread(job);
        Thread t2 = new Thread(job);
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println("Final count = " + c.getCount());    // always 2000
    }
}
```

Forms of Synchronization

Form	Syntax	Lock used
Synchronized instance method	public synchronized void m()	The current object (this)
Synchronized block	synchronized (obj) { ... }	The object obj given in parentheses
Static synchronized method	public static synchronized void m()	The Class object of the class

Synchronized block — lock only the critical part

```
public void increment() {
    // ... non-critical work here runs in parallel ...
    synchronized (this) {
        count++;                // only this part is protected
    }
}
```

Note: Synchronize as little code as possible. Locking large regions reduces the benefit of multithreading and increases the chance of deadlock.

7.6 Inter-Thread Communication

Synchronization prevents threads from interfering, but sometimes threads must also **cooperate** — one thread should wait until another has produced something. Java provides three methods in the class `java.lang.Object` for this purpose. They can be called **only from inside a synchronized method or block**; otherwise `IllegalMonitorStateException` is thrown.

Method	Action
<code>void wait()</code>	Releases the lock and puts the calling thread in the WAITING state until another thread notifies it
<code>void notify()</code>	Wakes up one thread waiting on this object's monitor
<code>void notifyAll()</code>	Wakes up all threads waiting on this object's monitor

Producer–Consumer Problem

The classic example: a **producer** thread puts items into a shared buffer and a **consumer** thread removes them. The producer must wait when the buffer is full, and the consumer must wait when it is empty.

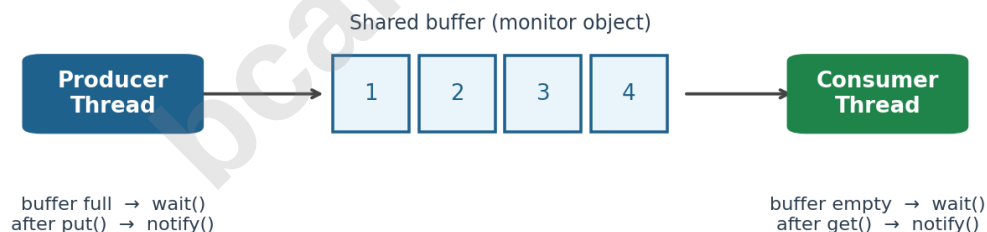


Figure 7.2 — Producer and consumer cooperating through `wait()` and `notify()`

Program 7.6 — Producer-Consumer with wait() and notify() (buffer of one item)

```
class Buffer {
    private int data;
    private boolean full = false;

    public synchronized void put(int v) throws InterruptedException {
        while (full) wait();           // buffer full -> producer waits
        data = v;
        full = true;
        System.out.println("Produced: " + v);
        notify();                      // wake the consumer
    }

    public synchronized int get() throws InterruptedException {
        while (!full) wait();          // buffer empty -> consumer waits
        full = false;
        System.out.println("Consumed: " + data);
        notify();                      // wake the producer
        return data;
    }
}
```

Program 7.6 (contd.) — Driver class

```
public class ProdConsDemo {
    public static void main(String[] args) {
        Buffer buf = new Buffer();
        Thread producer = new Thread(() -> {
            try { for (int i = 1; i <= 5; i++) buf.put(i); }
            catch (InterruptedException e) { }
        });
        Thread consumer = new Thread(() -> {
            try { for (int i = 1; i <= 5; i++) buf.get(); }
            catch (InterruptedException e) { }
        });
        producer.start();
        consumer.start();
    }
}
```

Output

Produced: 1 Consumed: 1 Produced: 2 Consumed: 2 ... up to 5.

The two threads strictly alternate because each waits for the other.

*Note: Always call wait() inside a **while** loop (not if), so the condition is re-checked after the thread wakes up. This guards against spurious wake-ups.*

sleep() versus wait()

Basis	sleep()	wait()
Defined in	Thread class (static)	Object class
Lock	Does not release the lock	Releases the lock
Wake-up	After the time expires	After notify()/notifyAll() (or timeout)
Called from	Anywhere	Only inside synchronized code
Purpose	Pause execution for some time	Inter-thread communication

7.7 Deadlock

A **deadlock** occurs when two or more threads are blocked forever, each waiting for a lock held by another thread in the group. None of them can proceed, so the program hangs.

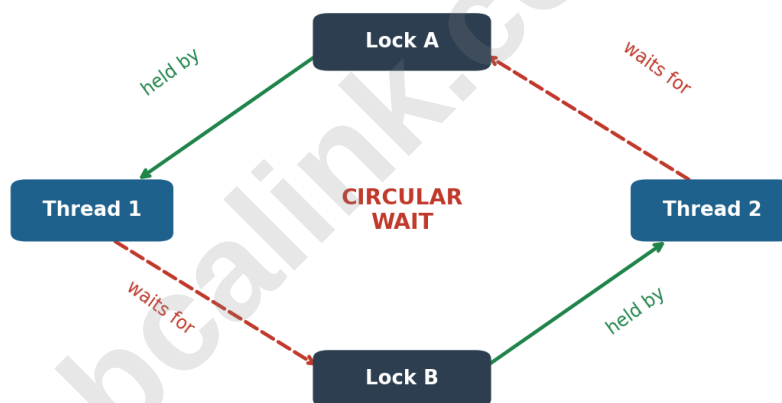


Figure 7.3 — Circular wait: Thread 1 holds Lock A and needs Lock B; Thread 2 holds Lock B and needs Lock A

Necessary Conditions for Deadlock

Deadlock can occur only if **all four** conditions hold at the same time (Coffman conditions):

- **Mutual exclusion** — a resource (lock) can be held by only one thread at a time.
- **Hold and wait** — a thread holds one lock while waiting for another.
- **No preemption** — a lock cannot be forcibly taken from a thread; it must be released voluntarily.
- **Circular wait** — a closed chain of threads exists where each waits for a lock held by the next.

Program 7.7 — A program that can deadlock

```
public class DeadlockDemo {
    static final Object lockA = new Object();
    static final Object lockB = new Object();

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized (lockA) {
                System.out.println("T1 holds A");
                try { Thread.sleep(100); } catch (InterruptedException e) { }
                synchronized (lockB) {                // T1 waits for B
                    System.out.println("T1 holds A and B");
                }
            }
        });
        Thread t2 = new Thread(() -> {
            synchronized (lockB) {
                System.out.println("T2 holds B");
                try { Thread.sleep(100); } catch (InterruptedException e) { }
                synchronized (lockA) {                // T2 waits for A
                    System.out.println("T2 holds A and B");
                }
            }
        });
        t1.start();
        t2.start();
    }
}
```

Result

Output stops after T1 holds A and T2 holds B. The program never ends — each thread waits forever for the lock the other one holds.

Prevention and Avoidance

- **Lock ordering** — make every thread acquire locks in the same fixed order (always A, then B). This breaks circular wait and is the most practical fix; in Program 7.7 it would remove the deadlock.
- **Lock only what is needed** — keep synchronized regions small and avoid nested locks.
- **Timeouts** — use `tryLock(time, unit)` from `java.util.concurrent.locks.ReentrantLock` so a thread gives up instead of waiting forever.
- **Avoid calling unknown code** (callbacks, overridable methods) while holding a lock.

*Note: Java does not detect or resolve deadlocks automatically. Tools such as `jstack` or a thread dump can reveal deadlocked threads. Related problems are **starvation** (a thread never gets CPU/lock time) and **livelock** (threads keep reacting to each other but make no progress).*

7.8 Likely Exam Questions

Type	Question
Short	Differentiate between process and thread.
Short	Why is implementing Runnable preferred over extending Thread?
Short	What is the difference between start() and run()?
Short	Differentiate between sleep() and wait().
Short	What is a race condition? How does synchronized prevent it?
Long	Explain the life cycle of a thread with a neat diagram.
Long	Describe the two ways of creating threads in Java with programs.
Long	Explain inter-thread communication. Write a program using wait() and notify().
Long	What is deadlock? State its conditions and show how it can be avoided.
Program	Write a program to create three threads with different priorities and display their names.

Key Takeaways

- ✓ A thread is a lightweight unit of execution; threads of a process share memory. Every program starts with the main thread.
- ✓ Create threads by **extending Thread** or **implementing Runnable** (preferred); always begin execution with **start()**, never **run()**.
- ✓ Thread states: NEW → RUNNABLE → (BLOCKED / WAITING / TIMED_WAITING) → TERMINATED.
- ✓ Priorities range from 1 to 10 (default 5) and are only hints to the scheduler.
- ✓ **synchronized** gives mutual exclusion using the object's monitor lock and prevents race conditions.
- ✓ **wait()**, **notify()**, **notifyAll()** enable communication between threads and work only inside synchronized code.
- ✓ Deadlock needs mutual exclusion, hold-and-wait, no preemption and circular wait; consistent lock ordering avoids it.

— End of Unit 7 —