Threads

Threads in Java; creating threads (Thread/Runnable), the thread lifecycle, thread methods, race conditions, synchronized, volatile, wait/notify, Atomic classes, ReentrantLock, and deadlock.

Advanced 55 min

Last updated: August 10, 2026
Read online: https://www.learnforgex.com/en/topics/threads

Threads

The seven topics so far (from Enum to Polymorphism) all dealt with code running on a single thread — we assumed the program was only ever in one place at a time. With this lesson we're entering a new **Concurrency** category: a world where multiple threads run at once, share state, and sometimes collide with each other. This first Concurrency lesson focuses on the core mechanics of the `Thread` class — creation, lifecycle, synchronization, `volatile`, locks, and deadlock; `ExecutorService`, `CompletableFuture`, and modern concurrency tools will be covered in a separate, later lesson.

What Is a Thread?

A **thread** is the smallest independently schedulable unit of execution in a program. A **process** — say, a running JVM instance — has its own memory, but the threads inside that process **share that same memory**; that sharing is both where a thread's power comes from and the root of most of the problems this lesson covers (race conditions, deadlock). Every Java program starts with at least one thread — the **main thread**, which runs the `main` method:

```
public class HelloThread {
    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()); // main
    }
}
```

When a program has more than one thread, that's called **multithreading** — multiple tasks within the same process can run **concurrently**, or (on a multi-core processor) genuinely **in parallel**.

Why Does It Exist?

A real-world example: think of a desktop application — while the user is downloading a file, you want the UI to stay responsive and keep reacting to mouse clicks. In a single-threaded program, **nothing** can run until the download finishes — the UI freezes. Multithreading moves the download work to a separate thread, freeing up the main thread (and therefore the UI). The same idea applies on the server side: a web server can serve hundreds of users **at once** by handling each incoming request on its own thread — instead of queuing them up and processing them one at a time.

History

The concept of a thread is much older than Java in the world of operating systems, but what set Java apart from many other languages was offering thread support as **part of the language and standard library itself**, from JDK 1.0 (1996) onward — the `Thread` class and the `synchronized` keyword have been there since day one. That early decision made Java attractive for server-side software of the era. But the earliest APIs (`wait` / `notify`, low-level `synchronized`) were hard to use correctly and error-prone; Java 5 (2004) brought much higher-level, safer tools with the `java.util.concurrent` package (`ExecutorService`, `ConcurrentHashMap`, the Atomic classes, and so on) — that package will be the focus of the next Concurrency lesson. Most recently, Java 21 (2023) fundamentally changed the cost of a thread with **virtual threads** (Project Loom) — a big enough development that it deserves its own future "Modern Concurrency" lesson.

Creating a Thread: Extending the Thread Class

The first of the two classic ways to create a thread is to extend the `Thread` class and override its `run()` method — a direct application of the mechanism we covered in the Inheritance lesson's "Method Overriding" section:

```
class GreetingThread extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": Hello from a thread!");
    }
}

class ExtendThreadExample {
    public static void main(String[] args) {
        GreetingThread thread = new GreetingThread();
        thread.start(); // runs run() on a NEW thread, not on main

        System.out.println(Thread.currentThread().getName() + ": Hello from main!");
    }
}
```

The code inside `run()` executes on a **new thread** once `start()` is called. The order of the output (whether `main` or the new thread prints first) is not guaranteed — it depends on the operating system's scheduler; that unpredictability is also the root cause of the problems we'll see in "Race Conditions." The critical difference between `start()` and calling `run()` directly is something we'll cover in detail in "Thread Methods: start(), join(), sleep(), interrupt()" — for now, just know that `start()` launches a new thread while `run()` is just an ordinary method call.

Creating a Thread: Implementing Runnable

The second, and usually preferred, way is to implement the `Runnable` interface and hand it to a `Thread`. Its advantage over extending `Thread` maps exactly onto the idea we covered in the Inheritance lesson's "Inheritance vs. Composition" section: a class that extends `Thread` can **never extend any other class**, because of Java's single inheritance restriction; a class that implements `Runnable` is free to extend whatever other class it needs:

```
class Task implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": running a task");
    }
}

class RunnableExample {
    public static void main(String[] args) throws InterruptedException {
        Task task = new Task();
        Thread thread = new Thread(task); // the Thread is GIVEN the work, it doesn't BE
the work
        thread.start();
        thread.join();

        // Modern style: a lambda, since Runnable has exactly one abstract method
        Thread lambdaThread = new Thread(() ->
            System.out.println(Thread.currentThread().getName() + ": running a lambda
task"));
        lambdaThread.start();
        lambdaThread.join();
    }
}
```

`Task` implements `Runnable`, but it isn't itself a `Thread` — it's **handed** to a `Thread` object via `new Thread(task)` and run from there, just like the Inheritance lesson's composition example handed an `Engine` to a `Car`. This distinction also makes the "is-a" vs. "has-a" difference concrete: a `Task` is not a `Thread`, it's merely the **work** a thread will run.

> In modern Java, `Runnable` implementations are usually written as a lambda: `new Thread(() -> System.out.println("running")).start();` — since `Runnable` has exactly one abstract method (`run()`), it's a **functional interface** (recall the Interface lesson's "Functional Interfaces and Lambdas" section).

Thread Lifecycle

Over its lifetime, a thread is always in one of the six states defined by the `Thread.State` enum (much like the fixed set of constants we saw in the Enum lesson, `Thread.State` is exactly that kind of fixed value set):

- **NEW:** The `Thread` object has been created, but `start()` hasn't been called yet.
- **RUNNABLE:** `start()` has been called; the thread is running or waiting for its turn on the CPU.
- **BLOCKED:** The thread is waiting for a lock held by another thread.
- **WAITING:** The thread is waiting indefinitely, via a call like `wait()` or `join()`.
- **TIMED_WAITING:** The thread is waiting for a bounded amount of time, via `sleep(ms)` or a timed `wait(ms)`.
- **TERMINATED:** The `run()` method has finished; the thread has ended.

```
class ThreadLifecycleExample {
    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> {
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        System.out.println("Before start: " + worker.getState()); // NEW

        worker.start();
        Thread.sleep(50); // give the worker time to enter sleep()
        System.out.println("While sleeping: " + worker.getState()); // TIMED_WAITING

        worker.join(); // main waits here until worker finishes
        System.out.println("After join: " + worker.getState()); // TERMINATED
    }
}
```

Notice that `worker.getState()` returns three different values at three different points: `NEW` before `start()` is called, `TIMED_WAITING` while it's sleeping, and `TERMINATED` after `join()` returns. `main`'s short `Thread.sleep(50)` here is just to guarantee the worker has actually entered its sleep by the time we check — in practice exact timing depends on the OS, but 50ms is more than enough to catch a 200ms sleep in progress.

Thread Methods: start(), join(), sleep(), interrupt()

There are four fundamental thread methods: `start()` launches a new thread (once more: calling `run()` directly does not open a new thread, it's just an ordinary method call); `join()` makes the calling thread (usually main) wait until the target thread finishes; `sleep(ms)` pauses the running thread for the given duration (it's a `static` method, and always puts the **calling** thread to sleep); `interrupt()` sends a thread a "cancellation request" — while the thread is blocked in a call like `sleep()`, `wait()`, or `join()`, that request reaches it as an `InterruptedException`.

```
class ThreadMethodsExample {
    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> {
            try {
                for (int i = 1; i <= 3; i++) {
                    System.out.println("working... " + i);
                    Thread.sleep(50);
                }
            } catch (InterruptedException e) {
                System.out.println("interrupted before finishing!");
                Thread.currentThread().interrupt(); // restore the interrupted status
            }
        });

        worker.start();
        worker.join(); // main blocks here until worker fully finishes
        System.out.println("worker is done, main continues");

        Thread another = new Thread(() -> {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("another was interrupted");
            }
        });
        another.start();
        another.interrupt(); // request cancellation while it's sleeping
        another.join();
    }
}
```

The `worker.join()` call **guarantees** that main won't print `"worker is done"` until `worker` has fully finished — without `join()`, that ordering would be left to chance. `another.interrupt()` wakes `another` up while it's inside `sleep(1000)` and throws an `InterruptedException` — this is the standard way to cancel a long-running operation from the outside.

> Swallowing `InterruptedException` in an empty `catch` block (`{}`) counts as a serious mistake — the interrupt signal is lost, and nothing calling your code ever learns the cancellation actually happened. Either rethrow the interrupt, or (as in the example) restore the state with `Thread.currentThread().interrupt()`.

Daemon Threads

Java can mark a thread as a **daemon** — the difference from a normal (non-daemon, or "user") thread is that the JVM **doesn't count daemon threads** when deciding whether to keep running: once every user thread has finished, the JVM shuts down even if daemon threads are still running:

```
class DaemonThreadExample {
    public static void main(String[] args) throws InterruptedException {
        Thread backgroundLogger = new Thread(() -> {
            int i = 0;
            while (true) {
                System.out.println("background log #" + i++);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    return;
                }
            }
        });

        backgroundLogger.setDaemon(true); // must be set BEFORE start()
        backgroundLogger.start();

        Thread.sleep(500); // main does a little "work"
        System.out.println("main is done -- JVM exits even though backgroundLogger never
finished");
    }
}
```

Because `backgroundLogger` is marked as a daemon thread, the JVM shuts down as soon as `main` finishes (after just 500ms) — even though `backgroundLogger`'s infinite loop never completes. `setDaemon(true)` **must** be called before `start()`; a thread's daemon status can't be changed once it's already running.

> The garbage collector's thread is a classic daemon thread built into the JDK itself — once the application's "real work" is done, you don't want a leftover GC task to keep the JVM alive.

Race Conditions

Now for the real problem: when two threads try to modify the same **shared state** at the same time, the result can be unpredictably and **unreproducibly** wrong. This is called a **race condition**:

```
class RaceConditionExample {
    static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        Runnable incrementTask = () -> {
            for (int i = 0; i < 100_000; i++) {
                counter++; // NOT atomic: read, increment, write -- three separate steps
            }
        };

        Thread t1 = new Thread(incrementTask);
        Thread t2 = new Thread(incrementTask);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Expected: 200000, Actual: " + counter); // usually LESS than
200000
    }
}
```

`counter++` looks like a single CPU operation but is actually three separate steps: read the value, increment it, write it back. If two threads interleave these three steps, one thread's increment can get **overwritten by the other's stale read** and simply vanish. Even if both threads increment the counter 100,000 times each, the result is almost never the expected 200,000 — and you'll likely see a **different** wrong number on every run, because exactly when each thread gets interrupted depends on the OS scheduler. We'll fix this problem in "Synchronization."

Synchronization

The `synchronized` keyword solves race conditions by guaranteeing that only **one** thread can be inside a given region at a time. Every Java object has an invisible lock (an **intrinsic lock**, or **monitor**); a thread entering a `synchronized` method or block acquires that lock and releases it on the way out — while the lock is held, no other thread can enter any region that needs the same lock:

```
class SafeCounter {
    private int count = 0;

    synchronized void increment() { // only one thread can be inside this method at a time
        count++;
    }

    int getCount() {
        return count;
    }
}

class SynchronizationExample {
    public static void main(String[] args) throws InterruptedException {
        SafeCounter counter = new SafeCounter();

        Runnable incrementTask = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(incrementTask);
        Thread t2 = new Thread(incrementTask);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Expected: 2000, Actual: " + counter.getCount()); // always 2000
    }
}
```

Because `increment()` is marked `synchronized`, while one thread is **inside** that method, no other thread can enter `increment()` on the same `SafeCounter` object — the three-step read-increment-write from "Race Conditions" is now effectively atomic. This time the result is always exactly the expected 2000.

> A `synchronized` block (`synchronized (lockObject) { ... }`) locks a **narrower** scope than a `synchronized` method — you lock only the lines that actually touch shared state, leaving the rest of the method (the parts that don't need the lock) free. Since less code stays locked, this usually performs better.

The volatile Keyword

`volatile` is often confused with `synchronized`, but it solves a completely different problem: **memory visibility**. For performance, each thread may keep its own CPU-core-local cached copy of a shared variable — which means that when one thread changes a value, other threads might **never see** that change. `volatile` guarantees that every read/write of a variable goes straight to main memory:

```
class VolatileExample {
    private static volatile boolean running = true;

    public static void main(String[] args) throws InterruptedException {
        Thread worker = new Thread(() -> {
            long iterations = 0;
            while (running) { // without volatile, this might loop forever
                iterations++;
            }
            System.out.println("worker stopped after seeing running = false, iterations=" +
iterations);
        });

        worker.start();
        Thread.sleep(100);
        running = false; // must be visible to the worker thread immediately
        worker.join();
        System.out.println("main confirmed worker stopped");
    }
}
```

If `running` weren't `volatile`, the JVM's compiler optimizations could cause the `worker` thread to read `running` only once and assume it stays true forever (never exiting the loop) — main's write of `running = false` might never become visible to the copy the worker sees. `volatile` guarantees that visibility.

> `volatile` only guarantees **visibility**, not **atomicity**. Writing `counter++` on a `volatile int counter` is still a race condition — the same three-step problem from "Race Conditions" applies exactly the same. For atomic increments you need either `synchronized` or the `AtomicInteger` we'll see in "Atomic Classes."

Thread Communication: wait(), notify(), notifyAll()

`synchronized` lets threads **block** each other; `wait()` / `notify()` / `notifyAll()` let threads **signal** each other — when a thread needs to wait until some condition becomes true (say, not consuming from an empty queue), it calls `wait()` to temporarily release its lock and sleep; another thread that changes that condition wakes it back up with `notify()` / `notifyAll()`:

```
class MessageBox {
    private String message;
    private boolean hasMessage = false;

    synchronized void put(String message) throws InterruptedException {
        while (hasMessage) {
            wait(); // release the lock and sleep until notified
        }
        this.message = message;
        hasMessage = true;
        notifyAll(); // wake up any thread waiting in take()
    }

    synchronized String take() throws InterruptedException {
        while (!hasMessage) {
            wait();
        }
        hasMessage = false;
        notifyAll(); // wake up any thread waiting in put()
        return message;
    }
}

class WaitNotifyExample {
    public static void main(String[] args) throws InterruptedException {
        MessageBox box = new MessageBox();

        Thread producer = new Thread(() -> {
            try {
                box.put("hello from producer");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        Thread consumer = new Thread(() -> {
            try {
                System.out.println("consumer received: " + box.take());
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        consumer.start();
        Thread.sleep(50); // let the consumer start waiting first
        producer.start();

        producer.join();
        consumer.join();
    }
}
```

Both `wait()` and `notify()` **must** be called inside a `synchronized` block/method — otherwise an `IllegalMonitorStateException` is thrown, since both need the same object's intrinsic lock (monitor). Calling `wait()` inside a `while` loop (never a single `if`) is also critical: when a

thread wakes up, it needs to re-check whether the condition is **still** true — to guard against a "spurious wakeup," where a thread can wake up without any `notify()` ever being called.

Modern Java codebases usually reach for the higher-level tools in `java.util.concurrent` (like `BlockingQueue`, `CountDownLatch`) instead of raw `wait()`/`notify()` — they achieve the same idea without falling into traps like spurious wakeups or holding the wrong lock; we'll cover those in the next Concurrency lesson.

Atomic Classes

Classes like `AtomicInteger`, `AtomicLong`, and `AtomicReference`, in the `java.util.concurrent.atomic` package, provide atomic (indivisible) operations without using `synchronized`. Internally they use a hardware-level **CAS** (compare-and-swap) operation: "if this variable's value is still X, set it to Y; otherwise retry" — a much lighter mechanism that doesn't require acquiring a lock:

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

class AtomicExample {
    static AtomicInteger counter = new AtomicInteger(0);

    public static void main(String[] args) throws InterruptedException {
        Runnable incrementTask = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.incrementAndGet(); // atomic read-increment-write, no synchronized
needed
            }
        };

        Thread t1 = new Thread(incrementTask);
        Thread t2 = new Thread(incrementTask);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Expected: 2000, Actual: " + counter.get()); // always 2000
    }
}
```

`AtomicInteger.incrementAndGet()` is the atomic counterpart to `counter++` from "Race Conditions" — the read, increment, and write happen as a single indivisible operation, with no `synchronized` block needed at all. For simple counters, flags, or reference updates, Atomic classes are usually lighter and faster than `synchronized` — but for complex, multi-step operations (like keeping several related decisions consistent, as in the bank account mini project coming up), `synchronized` or a `Lock` is usually the better tool.

Locks: ReentrantLock

`ReentrantLock`, in the `java.util.concurrent.locks` package, can do everything `synchronized` can, but offers more control: it provides flexibility `synchronized` doesn't, like **attempting** to acquire a lock (`tryLock()`) or waiting for a bounded amount of time:

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

class LockedCounter {
    private int count = 0;
    private final ReentrantLock lock = new ReentrantLock();

    void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock(); // MUST run even if the body throws
        }
    }

    int getCount() {
        return count;
    }
}

class ReentrantLockExample {
    public static void main(String[] args) throws InterruptedException {
        LockedCounter counter = new LockedCounter();

        Runnable incrementTask = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
        };

        Thread t1 = new Thread(incrementTask);
        Thread t2 = new Thread(incrementTask);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Expected: 2000, Actual: " + counter.getCount()); // always 2000

        // tryLock() -- give up immediately instead of blocking forever
        ReentrantLock another = new ReentrantLock();
        another.lock();
        try {
            Thread attempt = new Thread(() -> {
                if (another.tryLock()) {
                    System.out.println("acquired the lock");
                    another.unlock();
                } else {
                    System.out.println("lock was busy, giving up instead of blocking");
```

```
                }
            });
            attempt.start();
            attempt.join();
        } finally {
            another.unlock();
        }
    }
}
```

The code between `lock()` and `unlock()` can only be run by one thread at a time, just like a `synchronized` block — but `unlock()` **must** be called inside a `finally` block, or the lock is held forever if the code in between throws an exception (`synchronized` guarantees this release automatically, `ReentrantLock` doesn't). `tryLock()` lets a thread give up **immediately** and do something else if the lock is busy, instead of blocking forever waiting for it — something `synchronized` simply can't do.

> Calling `unlock()` outside a `finally` block, or not calling it at all, is the most common mistake with `ReentrantLock` — the lock is never released, and every thread waiting for it blocks forever. This produces the same outcome as one of the problems we'll see in "Deadlock."

Deadlock

A **deadlock** is when two (or more) threads block each other **forever**, each waiting for a lock the other one is holding. The classic scenario: if `Thread A` holds `lockA` and waits for `lockB`, while `Thread B` simultaneously holds `lockB` and waits for `lockA`, neither can ever make progress:

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

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            synchronized (lockA) {
                System.out.println("Thread 1: locked A, waiting for B");
                sleepQuietly(50);
                synchronized (lockB) {
                    System.out.println("Thread 1: locked B too");
                }
            }
        });

        Thread t2 = new Thread(() -> {
            synchronized (lockB) {
                System.out.println("Thread 2: locked B, waiting for A");
                sleepQuietly(50);
                synchronized (lockA) {
                    System.out.println("Thread 2: locked A too");
                }
            }
        });

        // Daemon so the JVM can still exit after main() returns, even though
        // a real deadlock would leave these two threads stuck forever.
        t1.setDaemon(true);
        t2.setDaemon(true);
        t1.start();
        t2.start();

        // A watchdog so this demo doesn't hang forever when you run it --
        // in a real deadlock there is no such safety net.
        t1.join(2000);
        t2.join(2000);
        if (t1.isAlive() || t2.isAlive()) {
            System.out.println("DEADLOCK DETECTED: both threads are still stuck after 2
seconds");
        }
    }

    static void sleepQuietly(long ms) {
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

`t1` locks `lockA` first and then waits for `lockB`, while `t2` locks `lockB` first and waits for `lockA` — both threads want the lock the other is holding, and neither ever gives up. The

most common way to prevent this is to guarantee that **every thread always acquires locks in the same order** — for instance, always locking the "smaller" one first; we'll come back to this rule in "Best Practices."

> In a real deadlock the program hangs forever — in this example, purely for teaching purposes, we add a "watchdog" (a `join` timeout) that **detects and reports** the deadlock instead; in real life there's no such safety net. Never attempt to acquire two locks in mismatched order like this in a real application.

Best Practices

- Always protect shared mutable state with `synchronized`, a `Lock`, or an Atomic class to guard against race conditions — don't assume "it's probably fine" without any of these (see "Race Conditions").
- Always acquire locks in the **same order** — most deadlocks come from different threads acquiring locks in a different order (see "Deadlock").
- Always call `wait()` inside a `while` loop, never a single `if` — to guard against spurious wakeups (see "Thread Communication: wait(), notify(), notifyAll()").
- If you're using `ReentrantLock`, always call `unlock()` inside a `finally` block (see the warning in "Locks: ReentrantLock").
- For a simple counter or flag, consider an Atomic class before reaching for `synchronized` — it's usually lighter and less error-prone (see "Atomic Classes").
- Avoid shared mutable state entirely where you can — immutable objects can never be subject to a race condition, because they can't be changed.

Common Mistakes

1. Calling `run()` instead of `start()`. Calling `run()` directly doesn't open a new thread — the code just runs on the calling thread, like an ordinary method call (see "Creating a Thread: Extending the Thread Class").

2. Swallowing `InterruptedException` in an empty `catch` block. The interrupt signal is lost, and whoever called your code never learns the cancellation happened (see the warning in "Thread Methods: start(), join(), sleep(), interrupt()").

3. Assuming `volatile` provides atomicity. `volatile` only guarantees visibility — `counter++` on a `volatile` counter is still a race condition (see the warning in "The volatile Keyword").

4. Calling `wait()` with an `if` instead of a `while`. To guard against spurious wakeups, the thread that wakes up needs to **re-check** the condition (see "Thread Communication: wait(), notify(), notifyAll()").

5. Acquiring locks in a different order in different threads. This is the most common cause of deadlocks — make sure every thread acquires locks in the same order (see "Deadlock").

6. Calling `ReentrantLock.unlock()` outside a `finally` block. If the locked region throws an exception, the lock is never released (see the warning in "Locks: ReentrantLock").

Summary, Cheat Sheet, and Glossary

Threads are Java's mechanism, present since JDK 1.0, for running multiple tasks concurrently within the same process. Key takeaways:

- There are two ways to create a thread: extend `Thread`, or (usually preferred) implement `Runnable`
- A thread is always in one of six states: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
- `start()` launches a new thread; calling `run()` directly does not
- `join()` waits for a thread to finish, `sleep()` pauses the running thread, `interrupt()` breaks a blocking call with an `InterruptedException`
- A race condition arises when multiple threads modify shared state without protecting it
- `synchronized` fixes race conditions by guaranteeing only one thread can enter a region at a time
- `volatile` provides memory visibility only, not atomicity
- `wait()` / `notify()` / `notifyAll()` let threads inside `synchronized` code signal each other; `wait()` must always be called inside a `while`
- Atomic classes (like `AtomicInteger`) provide lock-free atomic operations via CAS
- `ReentrantLock` offers extra flexibility over `synchronized` (like `tryLock()`), but requires manually calling `unlock()` in a `finally` block
- A deadlock is when threads block each other forever waiting on locks the other holds; always acquiring locks in the same order prevents it

Quick reference:

```
// Creating threads
Thread t1 = new Thread() {
    public void run() { /* ... */ }          // extends Thread
};
Thread t2 = new Thread(() -> { /* ... */ }); // implements Runnable (lambda)
t1.start(); // NOT t1.run()

// Basic methods
t1.join();          // wait for it to finish
Thread.sleep(100);  // pause the running thread
t1.interrupt();     // break a blocking call with InterruptedException

// Race condition -> fixed with synchronized
class Counter {
    private int count = 0;
    synchronized void increment() { count++; } // one thread at a time
}

// volatile -- visibility only
private volatile boolean running = true;

// Atomic -- lock-free atomic operation
AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet();

// ReentrantLock
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
    // ...
} finally {
    lock.unlock(); // always in finally
}

// Avoiding deadlock: always acquire locks in the same order
synchronized (lockA) {
    synchronized (lockB) {
        // ...
    }
}
```

Glossary

Thread — The smallest independently schedulable unit of execution in a program; threads within the same process share memory.

Process — A running instance of a program (say, a JVM instance) with its own memory; it hosts one or more threads.

Race condition — An unreproducible, unpredictable bug caused by multiple threads modifying shared state without protecting it.

`synchronized` — A keyword that guarantees only one thread can enter a region at a time, using that object's intrinsic lock (monitor).

`volatile` — A keyword that guarantees every read/write of a variable goes through main memory; provides visibility only, not atomicity.

Deadlock — Two or more threads blocking each other forever, each waiting on a lock the other one holds.

Daemon thread — A background thread the JVM doesn't count when deciding whether to keep running; the JVM shuts down once every user thread finishes, even if daemon threads are still active.

CAS (compare-and-swap) — The hardware-level operation Atomic classes use to make atomic updates without acquiring a lock.

`ReentrantLock` — An explicit locking tool in `java.util.concurrent.locks` that offers extra flexibility over `synchronized`, like `tryLock()` and timed waiting.

Appendix: Mini Project — A Thread-Safe Bank Account

In this mini project we carry the problem from "Race Conditions" into a realistic scenario: what happens when multiple threads try to withdraw from the same bank account **at the same time**, and how do we make that safe with `synchronized`:

```
class UnsafeBankAccount {
    private int balance;

    UnsafeBankAccount(int balance) {
        this.balance = balance;
    }

    void withdraw(int amount) {
        if (balance >= amount) { // check ...
            Thread.yield();      // widen the race window so the bug reliably shows up here
            balance -= amount;   // ... then act -- another thread can slip in between
        }
    }

    int getBalance() {
        return balance;
    }
}

class SafeBankAccount {
    private int balance;

    SafeBankAccount(int balance) {
        this.balance = balance;
    }

    synchronized void withdraw(int amount) {
        if (balance >= amount) { // check and act are now one atomic operation
            balance -= amount;
        }
    }

    synchronized int getBalance() {
        return balance;
    }
}
```

```
class BankAccountDemo {
    static void drainWithThreads(Runnable withdrawTask) throws InterruptedException {
        Thread t1 = new Thread(withdrawTask);
        Thread t2 = new Thread(withdrawTask);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
    }

    public static void main(String[] args) throws InterruptedException {
        UnsafeBankAccount unsafe = new UnsafeBankAccount(1000);
        drainWithThreads(() -> {
            for (int i = 0; i < 1000; i++) {
                unsafe.withdraw(1);
            }
        });
        // Starting balance 1000, two threads each attempt 1000 withdrawals of 1.
        // A correct implementation stops exactly at 0 -- the check-then-act race
        // here often lets both threads slip past the check, driving it BELOW zero.
        System.out.println("Unsafe balance: " + unsafe.getBalance());

        SafeBankAccount safe = new SafeBankAccount(1000);
        drainWithThreads(() -> {
            for (int i = 0; i < 1000; i++) {
                safe.withdraw(1);
            }
        });
        System.out.println("Safe balance: " + safe.getBalance()); // always exactly 0
    }
}
```

`UnsafeBankAccount.withdraw(...)` has no protection at all — when multiple threads withdraw at once, an update can get lost just like in "Race Conditions," and the account balance can drop to a **mathematically impossible** value (even going negative). `SafeBankAccount` makes `withdraw(...)` `synchronized`, so only one thread at a time can check and decrement the balance — the balance check and the update are now one indivisible operation.

> We didn't use `AtomicInteger` here instead of `synchronized`, because a withdrawal isn't a single-field update — it involves **both reading the balance and checking whether it's sufficient** (recall the last sentence of "Atomic Classes") — for this kind of "check, then act" operation, `synchronized` / `Lock` is usually the correct tool.

Appendix: Mini Project — Producer/Consumer

Our last mini project extends the idea from "Thread Communication: wait(), notify(), notifyAll()" into the classic **Producer/Consumer** problem: a shared queue with limited capacity, where the producer has to wait when it's full and the consumer has to wait when it's empty:

```
import java.util.LinkedList;
import java.util.Queue;

class SharedQueue {
    private final Queue<Integer> queue = new LinkedList<>();
    private final int capacity;

    SharedQueue(int capacity) {
        this.capacity = capacity;
    }

    synchronized void put(int value) throws InterruptedException {
        while (queue.size() == capacity) {
            wait(); // queue is full -- wait for the consumer to make room
        }
        queue.add(value);
        System.out.println("produced: " + value);
        notifyAll(); // wake up any consumer waiting in take()
    }

    synchronized int take() throws InterruptedException {
        while (queue.isEmpty()) {
            wait(); // queue is empty -- wait for the producer to add something
        }
        int value = queue.poll();
        System.out.println("consumed: " + value);
        notifyAll(); // wake up any producer waiting in put()
        return value;
    }
}
```

```
class ProducerConsumerDemo {
    public static void main(String[] args) throws InterruptedException {
        SharedQueue queue = new SharedQueue(3); // small capacity to force waiting on both
sides

        Thread producer = new Thread(() -> {
            try {
                for (int i = 1; i <= 6; i++) {
                    queue.put(i);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        Thread consumer = new Thread(() -> {
            try {
                for (int i = 1; i <= 6; i++) {
                    queue.take();
                    Thread.sleep(30); // consume a bit slower than it's produced
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        producer.start();
        consumer.start();
        producer.join();
        consumer.join();

        System.out.println("all 6 items produced and consumed");
    }
}
```

When `SharedQueue` is full, it makes the producer calling `put(...)` wait via `wait()`; when it's empty, it makes the consumer calling `take()` wait the same way — each side wakes back up via the other's `notifyAll()`. This is the single-message `MessageBox` idea from "Thread Communication: wait(), notify(), notifyAll()," expanded into the much more common real-world shape of a **bounded buffer**.

> In a real application you'd reach for a ready-made implementation of `java.util.concurrent.BlockingQueue` (like `ArrayBlockingQueue`) instead of writing this pattern from scratch with `wait()` / `notify()` — we'll cover that in the next Concurrency lesson. The point of writing it by hand here is to understand what's actually going on **inside** a `BlockingQueue`.