Chapter 10: Threads

Object Oriented Programming with Java

Threads and processes · Thread and Runnable · Life cycle · sleep · join · daemon · Stopping · Race conditions · synchronized · volatile · Lock · Deadlock · wait · notify · Condition · Synchronizers · Atomic classes · ThreadLocal · Thread pools

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_10_Threads.html

Learning Outcomes

Concepts

Explain the difference between a thread and a process, and between concurrency and parallelism.

Creation

Create and start threads by implementing `Runnable` or extending `Thread`; explain why `start()`, not `run()`.

Life cycle

Know the six states and trace the transitions caused by `sleep`, `join`, `wait` and lock waits.

Synchronization

Recognize race conditions; fix them with `synchronized`, `volatile`, `Lock` and atomic classes.

Waiting on conditions

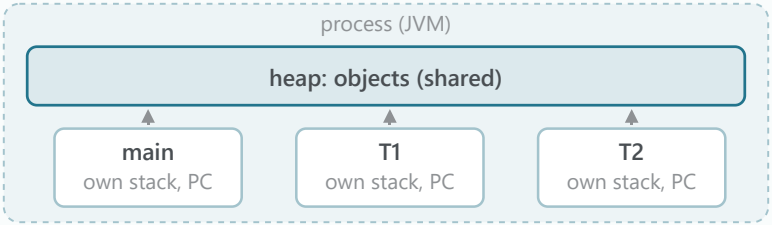
Write a producer–consumer with `wait/notify` and `Condition`; prevent deadlock.

Ready-made tools

Use `CountDownLatch`, `CyclicBarrier`, `Exchanger`, `Semaphore` and thread pools.

Threads

- A thread is the **basic unit of execution** in a Java program: an independent flow that runs instructions in order.
- Each thread is represented by an object of the `Thread` class.
- Threads are used for **multithreading** and **concurrent programming**.
- Every Java program starts with at least one thread: the **main thread**, which runs `main()`.



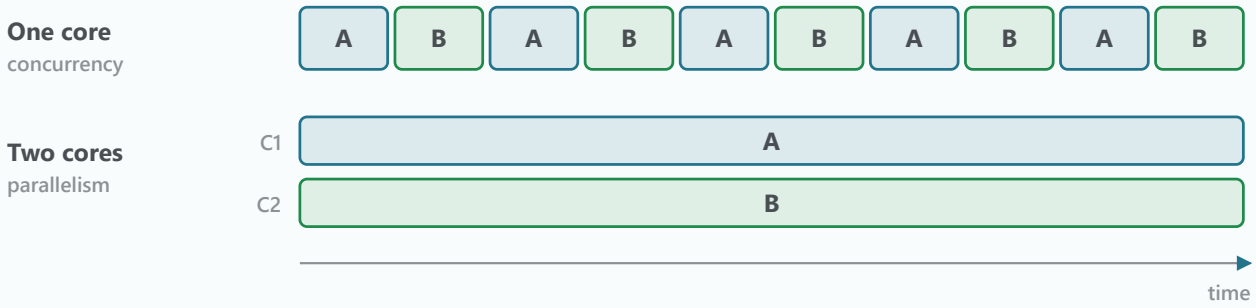
Analogy: a beehive

Each bee does its own job (a thread), but they all share the same hive and honeycomb (shared memory). If two bees try to fill the same cell at once, trouble follows: this chapter is about managing those "collisions".

Why?

Do background work without freezing the GUI, use a multicore CPU, keep working while waiting for the network or a file.

Concurrency and Parallelism



Concurrency

Several tasks **make progress in the same time span**. Possible even on one core: the OS runs threads in tiny alternating slices, so they seem to run at once.

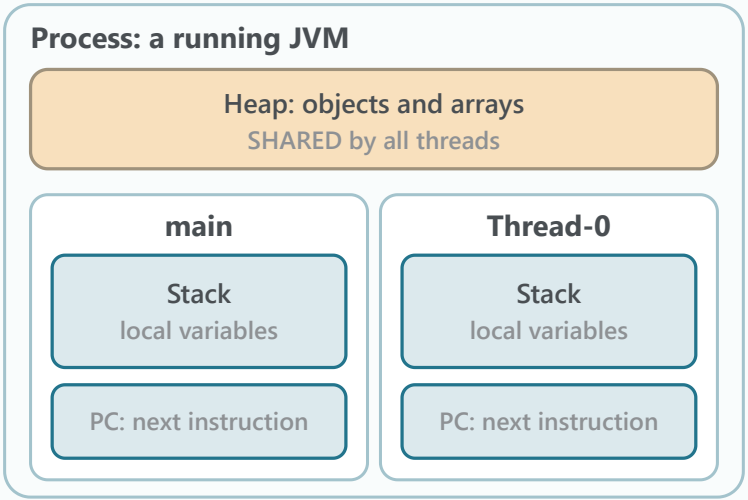
Parallelism

Tasks run **truly at the same time** on different cores. Requires more than one core; speeds up computation.

The **scheduler** of the JVM and the OS decides which thread runs when and on which core; the programmer cannot know this order.

BASIC CONCEPTS

Processes and Threads



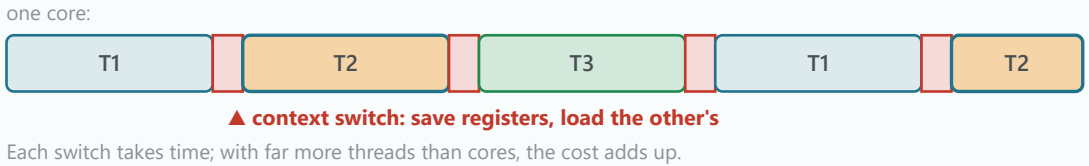
	Process	Thread
Memory	Own address space	Shares the process heap
Stack	—	Each has its own
Creation	Expensive	Cheap (lightweight process)
Communication	Files, sockets, pipes	Shared objects
Risk	Can't corrupt each other	Can corrupt shared data

A running Java program is **a process** with many threads inside. Objects live on the heap, so two threads can reach **the same object**; local variables live on each thread's own stack and are not shared.

BASIC CONCEPTS

Context Switching

- With more threads than cores, the OS runs them in turn using **time slices**.
- On each switch the running thread's state (program counter, registers, stack pointer) is **saved** and the next one's is **loaded**: this is a context switch.



Each switch takes time; with far more threads than cores, the cost adds up.

Cost

Each switch takes a few microseconds and invalidates part of the cache. Starting thousands of threads can **slow the program down**.

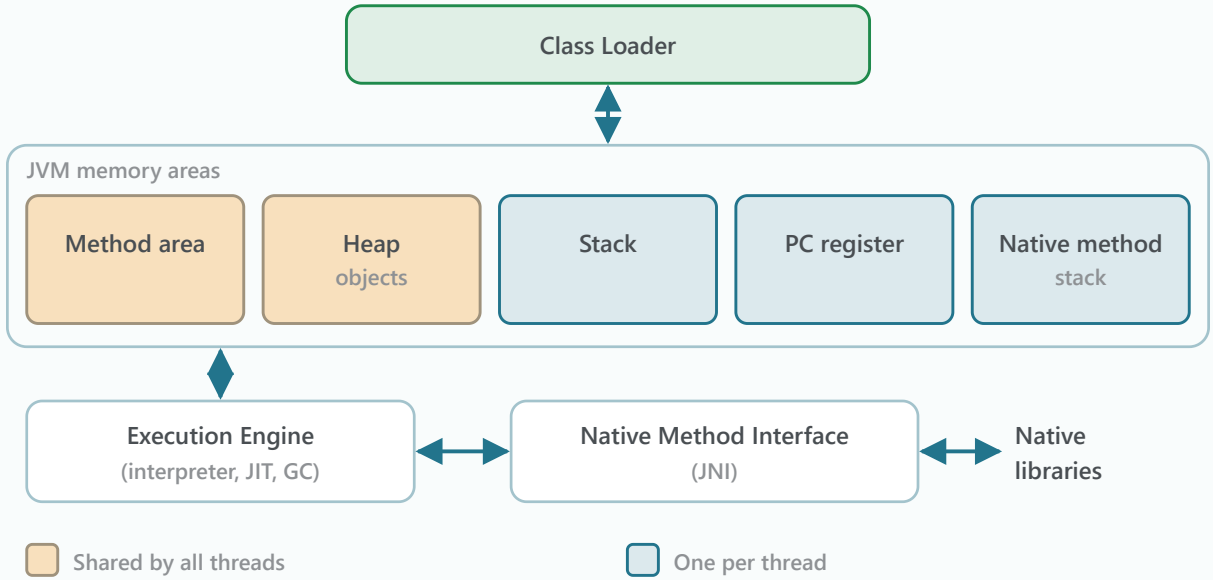
Rule of thumb

For CPU-bound work, threads ≈ cores (`Runtime.getRuntime().availableProcessors()`). For I/O-bound work (network, files) more can help; use a pool for that.

BASIC CONCEPTS

The Java Virtual Machine (JVM)

The JVM is a virtual environment that manages threads as **independent units of execution**.



When a thread is created, the JVM gives it its own stack, PC register and native method stack; the method area and the heap exist once.

The Thread Class

- Every thread is an object of the `Thread` class in `java.lang` (no import needed).
- `Thread` manages the thread's **life cycle** and provides the needed methods and properties.
- One approach: write a subclass of `Thread`.
- Override `run()` to define the thread's main logic.
- Call `start()` to start the thread.

Method	Purpose
`start()`	Starts a new thread, which runs `run()`
`run()`	The work the thread does
`sleep(ms)`	*static*: puts the current thread to sleep for a while
`join()`	Waits for that thread to finish
`yield()`	*static*: temporarily gives way to others
`interrupt()`	Sends an interrupt request
`currentThread()`	*static*: the thread running right now
`getName()`, `getState()`, `isAlive()`	Read information

Basic Attributes of a Thread (1)

Runnable
The object with the `run()` method, where the thread's main logic lives.

Name
The thread's name: `main`, `Thread-0`... Used in logs and for diagnosis. `getName()`, `setName()`

ID
A unique identifier, assigned **automatically** when the thread is created. `threadId()`

ThreadGroup
The group the thread belongs to; used to organize and manage threads.

Daemon
Serves other threads and runs periodic chores; the JVM **does not wait** for it when shutting down.

ContextClassLoader
The context class loader: the loader the thread uses to load classes dynamically.

These attributes are fields of the `Thread` object, with get/set methods to read and write them.

Basic Attributes of a Thread (2)

Priority

An integer from `MIN_PRIORITY` (1) to `MAX_PRIORITY` (10); default `NORM_PRIORITY` (5).

State

The current state: `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, `TERMINATED`.

Interrupted

A flag showing whether an interrupt was requested; set by `interrupt()`.

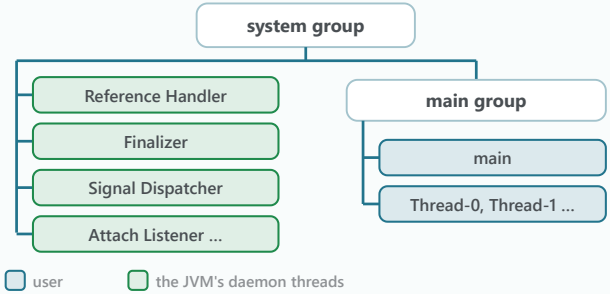
```
Thread t = Thread.currentThread();   // current
System.out.println("Name    : " + t.getName());
// threadId(): Java 19+ (getId() before)
System.out.println("ID      : " + t.threadId());
System.out.println("Priority: " + t.getPriority());
String group = t.getThreadGroup().getName();
System.out.println("Group   : " + group);
System.out.println("Daemon  : " + t.isDaemon());
System.out.println("State   : " + t.getState());
```

Output

```
Name    : main
ID      : 1
Priority: 5
Group   : main
Daemon  : false
State   : RUNNABLE
```

This snippet is inside `main`. `Thread.currentThread()` returns the thread running that line: here, the main thread.

Two Thread Groups at JVM Startup



Main thread group

Created at JVM startup; the main thread and the threads we create belong here by default.

System thread group

The JVM's own threads: reference handling, the Finalizer, the Signal Dispatcher…

LET'S LIST EVERY THREAD

```
Set<Thread> all =
        Thread.getAllStackTraces().keySet();
for (Thread t : all) {
    System.out.println(t.getThreadGroup().getName()
            + "  " + t.getName()
            + "  daemon=" + t.isDaemon());
}
```

Output (JDK 21, one run)

```
system  Notification Thread  daemon=true
main  main  daemon=false
system  Signal Dispatcher  daemon=true
system  Reference Handler  daemon=true
system  Attach Listener  daemon=true
InnocuousThreadGroup  Common-Cleaner  daemon=true
system  Finalizer  daemon=true
```

Notable Threads at JVM Startup (HotSpot)

Main Thread

The first thread to run when the JVM starts. `main()` runs here, and with it the core application logic.

Garbage Collector

Responsible for memory management: finds and frees objects with no references left, **automatically**. In HotSpot these are internal JVM threads that don't show up in the list.

Signal Dispatcher

Handles signals from the OS; for example, on a Ctrl+C shutdown request it starts an orderly JVM shutdown.

Finalizer

Calls objects' `finalize()` methods. `finalize()` was deprecated for removal in Java 18 (JEP 421); with `--finalization=disabled` it never starts. Use try-with-resources instead.

LIFE CYCLE

Thread States

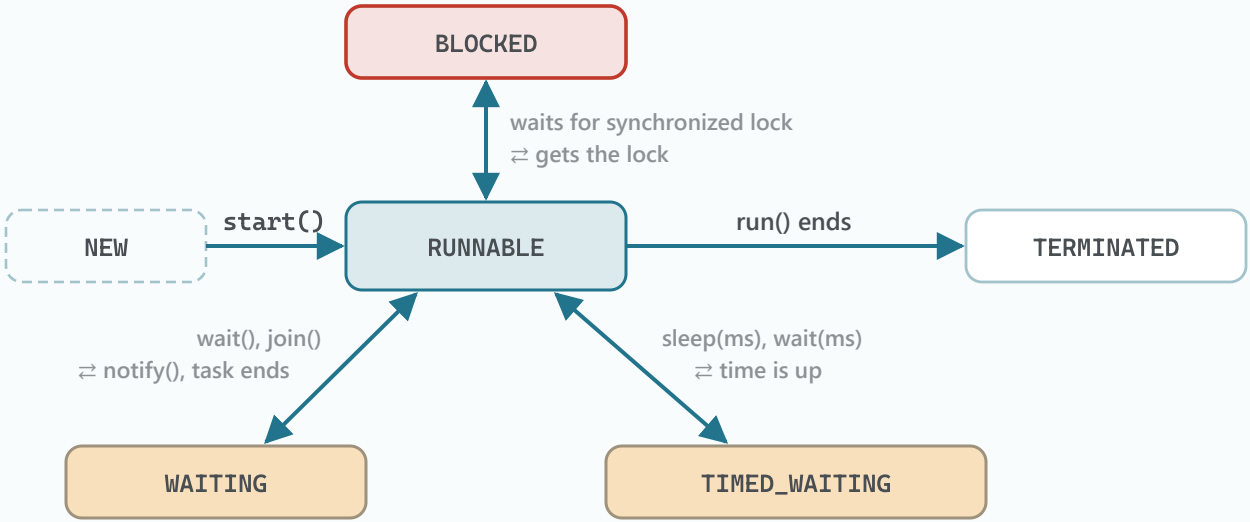
Threads pass through different states over their life cycle; `getState()` returns the state as a `Thread.State` constant.

State	Meaning	How you get there
`NEW`	Object created, not started yet	`new Thread(…)`; leaves on `start()`
`RUNNABLE`	Running or waiting for a CPU	After `start()`
`BLOCKED`	Waiting for a **monitor lock** to enter a `synchronized` block/method	While another thread holds the lock
`WAITING`	Waiting indefinitely; another thread must wake it	`Object.wait()`, `Thread.join()`, `LockSupport.park()`
`TIMED_WAITING`	Waiting with a timeout	`Thread.sleep(ms)`, `wait(ms)`, `join(ms)`, `parkNanos`, `parkUntil`
`TERMINATED`	Finished executing	When `run()` ends

Java has no separate “RUNNING” state; running threads and threads waiting for a CPU are both `RUNNABLE`.

LIFE CYCLE

State Diagram



One way

`NEW` → `RUNNABLE` → `TERMINATED`. A finished thread cannot be restarted.

Waiting states

Each of the three waiting states returns to `RUNNABLE`.

The difference

`BLOCKED`: for a lock;
`WAITING`: for an event;
`TIMED_WAITING`: with a time limit.

Observing the States, Step by Step

```
1  public static void main(String[] args) {
▶      Thread sleeper = new Thread(() -> pause(1000));
3      System.out.println("1) " + sleeper.getState());
4      sleeper.start();
5      pause(100);
6      System.out.println("2) " + sleeper.getState());
7      Thread waiter = new Thread(() -> waitFor(sleeper));
8      waiter.start();
9      pause(100);
10     System.out.println("3) " + waiter.getState());
11     Thread blocked = new Thread(() -> {
12         synchronized (lock) { }
13     });
14     synchronized (lock) {
15         blocked.start();
16         pause(100);
17         System.out.println("4) " + blocked.getState());
18     }
19     waitFor(waiter);
20     System.out.println("5) " + sleeper.getState());
21 }
```

main	RUNNABLE
sleeper	NEW
waiter	not yet
blocked	not yet

`sleeper` is created but not started: `NEW`.

Step 1 / 11

OUTPUT

`pause(ms)` and `waitFor(t)` are helpers that wrap `Thread.sleep` and `t.join()` in try-catch. The pauses make the output always come out like this.

Observing the States, Step by Step

```
1  public static void main(String[] args) {
2      Thread sleeper = new Thread(() -> pause(1000));
▶      System.out.println("1) " + sleeper.getState());
4      sleeper.start();
5      pause(100);
6      System.out.println("2) " + sleeper.getState());
7      Thread waiter = new Thread(() -> waitFor(sleeper));
8      waiter.start();
9      pause(100);
10     System.out.println("3) " + waiter.getState());
11     Thread blocked = new Thread(() -> {
12         synchronized (lock) { }
13     });
14     synchronized (lock) {
15         blocked.start();
16         pause(100);
17         System.out.println("4) " + blocked.getState());
18     }
19     waitFor(waiter);
20     System.out.println("5) " + sleeper.getState());
21 }
```

main	RUNNABLE
sleeper	NEW
waiter	not yet
blocked	not yet

The state is printed.

Step 2 / 11

OUTPUT

```
1) NEW
```

`pause(ms)` and `waitFor(t)` are helpers that wrap `Thread.sleep` and `t.join()` in try-catch. The pauses make the output always come out like this.

Observing the States, Step by Step

```
1  public static void main(String[] args) {
2      Thread sleeper = new Thread(() -> pause(1000));
3      System.out.println("1) " + sleeper.getState());
▶      sleeper.start();
5      pause(100);
6      System.out.println("2) " + sleeper.getState());
7      Thread waiter = new Thread(() -> waitFor(sleeper));
8      waiter.start();
9      pause(100);
10     System.out.println("3) " + waiter.getState());
11     Thread blocked = new Thread(() -> {
12         synchronized (lock) { }
13     });
14     synchronized (lock) {
15         blocked.start();
16         pause(100);
17         System.out.println("4) " + blocked.getState());
18     }
19     waitFor(waiter);
20     System.out.println("5) " + sleeper.getState());
21 }
```

main	RUNNABLE
sleeper	RUNNABLE
waiter	not yet
blocked	not yet

`start()` : the new thread is ready to run.

Step 3 / 11

OUTPUT

```
1) NEW
```

`pause(ms)` and `waitFor(t)` are helpers that wrap `Thread.sleep` and `t.join()` in try-catch. The pauses make the output always come out like this.

Observing the States, Step by Step

```
1  public static void main(String[] args) {
2      Thread sleeper = new Thread(() -> pause(1000));
3      System.out.println("1) " + sleeper.getState());
4      sleeper.start();
▶      pause(100);
6      System.out.println("2) " + sleeper.getState());
7      Thread waiter = new Thread(() -> waitFor(sleeper));
8      waiter.start();
9      pause(100);
10     System.out.println("3) " + waiter.getState());
11     Thread blocked = new Thread(() -> {
12         synchronized (lock) { }
13     });
14     synchronized (lock) {
15         blocked.start();
16         pause(100);
17         System.out.println("4) " + blocked.getState());
18     }
19     waitFor(waiter);
20     System.out.println("5) " + sleeper.getState());
21 }
```

main	TIMED_WAITING
sleeper	TIMED_WAITING
waiter	not yet
blocked	not yet

Both are in `sleep` : main for 100 ms, sleeper for 1000 ms.

Step 4 / 11

OUTPUT

```
1) NEW
```

`pause(ms)` and `waitFor(t)` are helpers that wrap `Thread.sleep` and `t.join()` in try-catch. The pauses make the output always come out like this.

Observing the States, Step by Step

```
1  public static void main(String[] args) {
2      Thread sleeper = new Thread(() -> pause(1000));
3      System.out.println("1) " + sleeper.getState());
4      sleeper.start();
5      pause(100);
▶      System.out.println("2) " + sleeper.getState());
7      Thread waiter = new Thread(() -> waitFor(sleeper));
8      waiter.start();
9      pause(100);
10     System.out.println("3) " + waiter.getState());
11     Thread blocked = new Thread(() -> {
12         synchronized (lock) { }
13     });
14     synchronized (lock) {
15         blocked.start();
16         pause(100);
17         System.out.println("4) " + blocked.getState());
18     }
19     waitFor(waiter);
20     System.out.println("5) " + sleeper.getState());
21 }
```

main	RUNNABLE
sleeper	TIMED_WAITING
waiter	not yet
blocked	not yet

main woke up; sleeper is still asleep: `TIMED_WAITING`.

Step 5 / 11

OUTPUT

```
1) NEW
2) TIMED_WAITING
```

`pause(ms)` and `waitFor(t)` are helpers that wrap `Thread.sleep` and `t.join()` in try-catch. The pauses make the output always come out like this.

Observing the States, Step by Step

```
1  public static void main(String[] args) {
2      Thread sleeper = new Thread(() -> pause(1000));
3      System.out.println("1) " + sleeper.getState());
4      sleeper.start();
5      pause(100);
6      System.out.println("2) " + sleeper.getState());
7      Thread waiter = new Thread(() -> waitFor(sleeper));
▶      waiter.start();
9      pause(100);
10     System.out.println("3) " + waiter.getState());
11     Thread blocked = new Thread(() -> {
12         synchronized (lock) { }
13     });
14     synchronized (lock) {
15         blocked.start();
16         pause(100);
17         System.out.println("4) " + blocked.getState());
18     }
19     waitFor(waiter);
20     System.out.println("5) " + sleeper.getState());
21 }
```

main	RUNNABLE
sleeper	TIMED_WAITING
waiter	WAITING
blocked	not yet

`waiter` calls `sleeper.join()`: it waits indefinitely, `WAITING`.

Step 6 / 11

OUTPUT

```
1) NEW
2) TIMED_WAITING
```

`pause(ms)` and `waitFor(t)` are helpers that wrap `Thread.sleep` and `t.join()` in try-catch. The pauses make the output always come out like this.

Observing the States, Step by Step

```
1  public static void main(String[] args) {
2      Thread sleeper = new Thread(() -> pause(1000));
3      System.out.println("1) " + sleeper.getState());
4      sleeper.start();
5      pause(100);
6      System.out.println("2) " + sleeper.getState());
7      Thread waiter = new Thread(() -> waitFor(sleeper));
8      waiter.start();
9      pause(100);
▶      System.out.println("3) " + waiter.getState());
11     Thread blocked = new Thread(() -> {
12         synchronized (lock) { }
13     });
14     synchronized (lock) {
15         blocked.start();
16         pause(100);
17         System.out.println("4) " + blocked.getState());
18     }
19     waitFor(waiter);
20     System.out.println("5) " + sleeper.getState());
21 }
```

main	RUNNABLE
sleeper	TIMED_WAITING
waiter	WAITING
blocked	not yet

After 100 ms the state is printed.

Step 7 / 11

OUTPUT

```
1) NEW
2) TIMED_WAITING
3) WAITING
```

`pause(ms)` and `waitFor(t)` are helpers that wrap `Thread.sleep` and `t.join()` in try-catch. The pauses make the output always come out like this.

Observing the States, Step by Step

```
1  public static void main(String[] args) {
2      Thread sleeper = new Thread(() -> pause(1000));
3      System.out.println("1) " + sleeper.getState());
4      sleeper.start();
5      pause(100);
6      System.out.println("2) " + sleeper.getState());
7      Thread waiter = new Thread(() -> waitFor(sleeper));
8      waiter.start();
9      pause(100);
10     System.out.println("3) " + waiter.getState());
11     Thread blocked = new Thread(() -> {
12         synchronized (lock) { }
13     });
14     synchronized (lock) {
▶          blocked.start();
16         pause(100);
17         System.out.println("4) " + blocked.getState());
18     }
19     waitFor(waiter);
20     System.out.println("5) " + sleeper.getState());
21 }
```

main	RUNNABLE
sleeper	TIMED_WAITING
waiter	WAITING
blocked	BLOCKED

main holds `lock` ; `blocked` wants the same lock: `BLOCKED` .

Step 8 / 11

OUTPUT

```
1) NEW
2) TIMED_WAITING
3) WAITING
```

`pause(ms)` and `waitFor(t)` are helpers that wrap `Thread.sleep` and `t.join()` in try-catch. The pauses make the output always come out like this.

Observing the States, Step by Step

```
1  public static void main(String[] args) {
2      Thread sleeper = new Thread(() -> pause(1000));
3      System.out.println("1) " + sleeper.getState());
4      sleeper.start();
5      pause(100);
6      System.out.println("2) " + sleeper.getState());
7      Thread waiter = new Thread(() -> waitFor(sleeper));
8      waiter.start();
9      pause(100);
10     System.out.println("3) " + waiter.getState());
11     Thread blocked = new Thread(() -> {
12         synchronized (lock) { }
13     });
14     synchronized (lock) {
15         blocked.start();
16         pause(100);
▶          System.out.println("4) " + blocked.getState());
18     }
19     waitFor(waiter);
20     System.out.println("5) " + sleeper.getState());
21 }
```

main	RUNNABLE
sleeper	TIMED_WAITING
waiter	WAITING
blocked	BLOCKED

main still holds the lock; the state is printed.

Step 9 / 11

OUTPUT

```
1) NEW
2) TIMED_WAITING
3) WAITING
4) BLOCKED
```

`pause(ms)` and `waitFor(t)` are helpers that wrap `Thread.sleep` and `t.join()` in try-catch. The pauses make the output always come out like this.

Observing the States, Step by Step

```
1  public static void main(String[] args) {
2      Thread sleeper = new Thread(() -> pause(1000));
3      System.out.println("1) " + sleeper.getState());
4      sleeper.start();
5      pause(100);
6      System.out.println("2) " + sleeper.getState());
7      Thread waiter = new Thread(() -> waitFor(sleeper));
8      waiter.start();
9      pause(100);
10     System.out.println("3) " + waiter.getState());
11     Thread blocked = new Thread(() -> {
12         synchronized (lock) { }
13     });
14     synchronized (lock) {
15         blocked.start();
16         pause(100);
17         System.out.println("4) " + blocked.getState());
18     }
▶      waitFor(waiter);
20     System.out.println("5) " + sleeper.getState());
21 }
```

main	WAITING
sleeper	TERMINATED
waiter	TERMINATED
blocked	TERMINATED

When the block ends the lock is released, and blocked enters and finishes. main waits for `waiter`; sleeper ends after its 1 s, then waiter ends too.

Step 10 / 11

OUTPUT

```
1) NEW
2) TIMED_WAITING
3) WAITING
4) BLOCKED
```

`pause(ms)` and `waitFor(t)` are helpers that wrap `Thread.sleep` and `t.join()` in try-catch. The pauses make the output always come out like this.

Observing the States, Step by Step

```
1  public static void main(String[] args) {
2      Thread sleeper = new Thread(() -> pause(1000));
3      System.out.println("1) " + sleeper.getState());
4      sleeper.start();
5      pause(100);
6      System.out.println("2) " + sleeper.getState());
7      Thread waiter = new Thread(() -> waitFor(sleeper));
8      waiter.start();
9      pause(100);
10     System.out.println("3) " + waiter.getState());
11     Thread blocked = new Thread(() -> {
12         synchronized (lock) { }
13     });
14     synchronized (lock) {
15         blocked.start();
16         pause(100);
17         System.out.println("4) " + blocked.getState());
18     }
19     waitFor(waiter);
▶      System.out.println("5) " + sleeper.getState());
21 }
```

main	RUNNABLE
sleeper	TERMINATED
waiter	TERMINATED
blocked	TERMINATED

A thread whose `run()` has ended: `TERMINATED`.

Step 11 / 11

OUTPUT

```
1) NEW
2) TIMED_WAITING
3) WAITING
4) BLOCKED
5) TERMINATED
```

`pause(ms)` and `waitFor(t)` are helpers that wrap `Thread.sleep` and `t.join()` in try-catch. The pauses make the output always come out like this.

The Runnable Interface

- Every thread has a `Runnable` object that it runs.
- The `Runnable` interface has just one method, `run()`.
- A class that implements it defines `run()`: the thread's work.
- Since it is a functional interface (a single abstract method), Java 8+ also lets you write a lambda.

DEFINITION IN THE JDK

```
package java.lang;

public interface Runnable {
    void run();
}
```

AN IMPLEMENTING CLASS

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // The thread's tasks
    }
}
```

JAVA 8+: A LAMBDA

```
Runnable r = () -> System.out.println("hello");
```

Starting a Thread

```
public class Worker implements Runnable {
    // The thread's work goes here
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println("Worker running: " + name);
    }
    public static void main(String[] args) {
        // Start the thread
        Worker w = new Worker();
        Thread t = new Thread(w);
        t.start();
        System.out.println("main goes on");
    }
}
```

- The `Runnable` object is passed to the `Thread` constructor.
- Calling `start()` starts the thread.
- Don't call `run()` directly: the code would then run on the caller's thread.

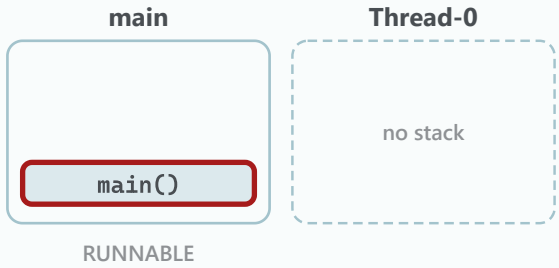
Output (a real run)

```
main goes on
Worker running: Thread-0
```

The order of the lines **may change** from run to run: main carries on as soon as `start()` returns; the scheduler decides when the new thread runs.

start(): Two Stacks, Two Flows

```
1  public class Worker implements Runnable {
2      // The thread's work goes here
3      public void run() {
4          String name = Thread.currentThread().getName();
5          System.out.println("Worker running: " + name);
6      }
7      public static void main(String[] args) {
8          // Start the thread
▶          Worker w = new Worker();
10         Thread t = new Thread(w);
11         t.start();
12         System.out.println("main goes on");
13     }
14 }
```



main is running; a `Worker` object is created on the heap. There is still only one thread.

Step 1 / 8

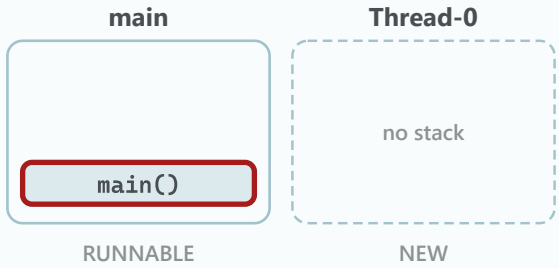
VARIABLES

`w = Worker object` `t = –`

OUTPUT

start(): Two Stacks, Two Flows

```
1  public class Worker implements Runnable {
2      // The thread's work goes here
3      public void run() {
4          String name = Thread.currentThread().getName();
5          System.out.println("Worker running: " + name);
6      }
7      public static void main(String[] args) {
8          // Start the thread
9          Worker w = new Worker();
           Thread t = new Thread(w);
11         t.start();
12         System.out.println("main goes on");
13     }
14 }
```



The `Thread` object exists (`NEW`) but has no separate flow or stack yet.

Step 2 / 8

VARIABLES

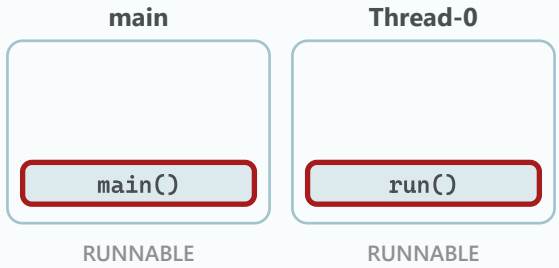
`w = Worker object`

`t = Thread-0 (NEW)`

OUTPUT

start(): Two Stacks, Two Flows

```
1  public class Worker implements Runnable {
2      // The thread's work goes here
3      public void run() {
4          String name = Thread.currentThread().getName();
5          System.out.println("Worker running: " + name);
6      }
7      public static void main(String[] args) {
8          // Start the thread
9          Worker w = new Worker();
10         Thread t = new Thread(w);
▶          t.start();
12         System.out.println("main goes on");
13     }
14 }
```



`start()` : the JVM creates a new stack and starts `run()` on it. `start()` returns at once.

Step 3 / 8

VARIABLES

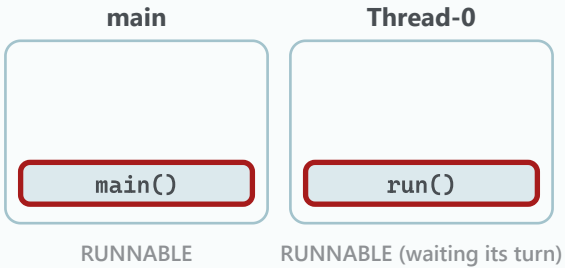
`w = Worker object`

`t = Thread-0 (RUNNABLE)`

OUTPUT

start(): Two Stacks, Two Flows

```
1  public class Worker implements Runnable {
2      // The thread's work goes here
3      public void run() {
4          String name = Thread.currentThread().getName();
5          System.out.println("Worker running: " + name);
6      }
7      public static void main(String[] args) {
8          // Start the thread
9          Worker w = new Worker();
10         Thread t = new Thread(w);
11         t.start();
▶          System.out.println("main goes on");
13     }
14 }
```



One possible order: main moves first and prints.

Step 4 / 8

VARIABLES

`w = Worker object`

`t = Thread-0 (RUNNABLE)`

OUTPUT

```
main goes on
```

start(): Two Stacks, Two Flows

```
1  public class Worker implements Runnable {
2      // The thread's work goes here
3      public void run() {
4          String name = Thread.currentThread().getName();
5          System.out.println("Worker running: " + name);
6      }
7      public static void main(String[] args) {
8          // Start the thread
9          Worker w = new Worker();
10         Thread t = new Thread(w);
11         t.start();
12         System.out.println("main goes on");
▶      }
14 }
```



`main()` has ended but the program hasn't: the JVM waits until all non-daemon threads finish.

Step 5 / 8

VARIABLES

`t = Thread-0 (RUNNABLE)`

OUTPUT

```
main goes on
```

start(): Two Stacks, Two Flows

```
1  public class Worker implements Runnable {
2      // The thread's work goes here
3      public void run() {
4          String name = Thread.currentThread().getName();
5          System.out.println("Worker running: " + name);
6      }
7      public static void main(String[] args) {
8          // Start the thread
9          Worker w = new Worker();
10         Thread t = new Thread(w);
11         t.start();
12         System.out.println("main goes on");
13     }
14 }
```



Thread-0 runs on its own stack; `name` is its local variable.

Step 6 / 8

VARIABLES

`t = Thread-0 (RUNNABLE)`

`name = "Thread-0"`

OUTPUT

```
main goes on
```

start(): Two Stacks, Two Flows

```
1  public class Worker implements Runnable {
2      // The thread's work goes here
3      public void run() {
4          String name = Thread.currentThread().getName();
5          System.out.println("Worker running: " + name);
6      }
7      public static void main(String[] args) {
8          // Start the thread
9          Worker w = new Worker();
10         Thread t = new Thread(w);
11         t.start();
12         System.out.println("main goes on");
13     }
14 }
```



Thread-0 prints.

VARIABLES

`t = Thread-0 (RUNNABLE)`

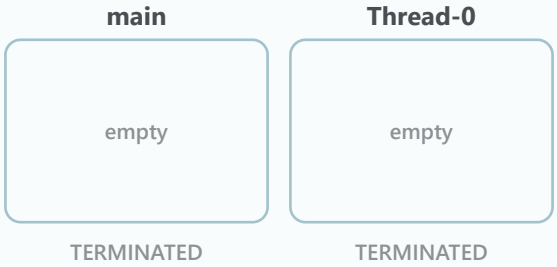
`name = "Thread-0"`

OUTPUT

```
main goes on
Worker running: Thread-0
```

start(): Two Stacks, Two Flows

```
1  public class Worker implements Runnable {
2      // The thread's work goes here
3      public void run() {
4          String name = Thread.currentThread().getName();
5          System.out.println("Worker running: " + name);
6      }
7      public static void main(String[] args) {
8          // Start the thread
9          Worker w = new Worker();
10         Thread t = new Thread(w);
11         t.start();
12         System.out.println("main goes on");
13     }
14 }
```



`run()` is done: Thread-0 is `TERMINATED`. No threads are left running, so the JVM exits.

Step 8 / 8

VARIABLES

`t = Thread-0 (TERMINATED)`

OUTPUT

```
main goes on
Worker running: Thread-0
```

Extending the Thread Class

```
public class Worker2 extends Thread {
    // The thread's work goes here
    public void run() {
        String name = getName();   // inherited from Thread
        System.out.println("Worker running: " + name);
    }
    public static void main(String[] args) {
        // Create and start the thread
        Worker2 w = new Worker2();
        w.start();
        System.out.println("main goes on");
    }
}
```

- A thread can also be created by extending the `Thread` class.
- Instead of implementing `Runnable`, write a subclass of `Thread`; `Thread` already implements `Runnable`.
- The object itself is a `Thread`, so you call `w.start()` and `getName()` directly.

Output (a real run)

```
main goes on
Worker running: Thread-0
```

Again, the order may change.

start() or run()?

```java
public class StartVsRun {
    public static void main(String[] args)
            throws InterruptedException {
        Thread t = new Thread(() -> {
            String name = Thread.currentThread().getName();
            System.out.println("inside run(): " + name);
        });
        t.run();     // WRONG: main runs it
        t.start();   // RIGHT: a new thread
        t.join();
    }
}
```

Output

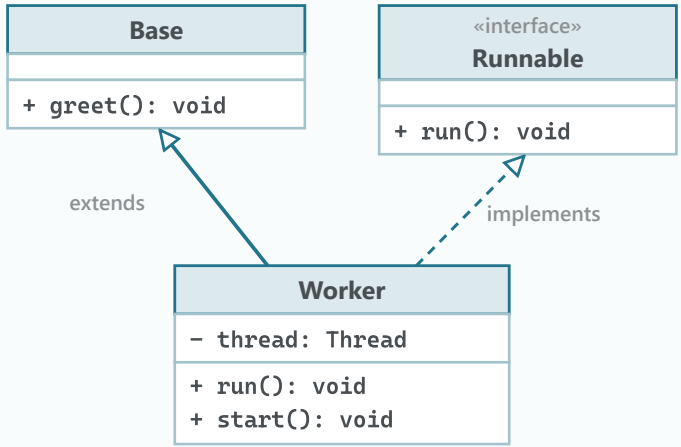
```
inside run(): main
inside run(): Thread-0
```

Call	What happens
`t.run()`	An ordinary method call: runs on the **same** thread (main), no new stack; main waits until it ends.
`t.start()`	The JVM creates a **new** thread and stack; `run()` runs there and `start()` returns at once.
`t.start()` a second time	`IllegalThreadStateException` : a `Thread` object can be started only once.

Rule: put the thread's **work** in `run()` ; call `start()` to **start** the thread.

The Single Inheritance Problem

- Java lets a class inherit from **only one class**.
- If the runnable class already extends another class (`Base`), it **cannot extend** `Thread` : `extends Base, Thread` won't compile.
- The fix: the class implements `Runnable` and keeps a field of type `Thread` .
- There is no limit on interfaces, which is why `Runnable` is usually preferred.



Single Inheritance: The Fix in Code

```
class Base {
    public void greet() {
        System.out.println("Hello from Base");
    }
}
class Worker extends Base implements Runnable {
    private Thread thread;   // Thread as a field
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println("run(): " + name);
    }
    public void start() {
        thread = new Thread(this);
        thread.start();
    }
}
public class SingleInheritance {
    public static void main(String[] args) {
        Worker w = new Worker();
        w.greet();   // inherited from Base
        w.start();   // our own start()
    }
}
```

Output

```
Hello from Base
run(): Thread-0
```

- `Worker` inherits `Base`'s methods and can also run on a thread.
- Our own `start()` hands itself to a `Thread` with `new Thread(this)`.
- `this` is a `Runnable`, which the `Thread` constructor accepts.

Three Ways: A Comparison

```
public class ThreeWays {
    static class Task implements Runnable {     // 1)
        public void run() {
            System.out.println("1) Runnable class");
        }
    }
    static class MyThread extends Thread {       // 2)
        public void run() {
            System.out.println("2) Thread subclass");
        }
    }
    public static void main(String[] args)
            throws InterruptedException {
        Thread t1 = new Thread(new Task());
        Thread t2 = new MyThread();
        // 3) Lambda (Java 8+)
        Thread t3 = new Thread(
                () -> System.out.println("3) Lambda"));
        t1.start(); t1.join();
        t2.start(); t2.join();
        t3.start(); t3.join();
    }
}
```

Output

```
1) Runnable class
2) Thread subclass
3) Lambda
```

Way	When?
`implements Runnable`	The class already extends another; keeps the task separate from the thread
`extends Thread`	Short, standalone examples; uses up the one superclass
Lambda (Java 8+)	The handiest way for short tasks

Each `start()` is followed right away by `join()`, so the output order is fixed. `static class`: a class nested inside another class.

Two Threads: One Possible Order

```
1  public class TwoThreads {
2      static void task() {
3          String name = Thread.currentThread().getName();
4          for (int i = 1; i <= 2; i++) {
5              System.out.println(name + ": step " + i);
6          }
7      }
8      public static void main(String[] args)
9              throws InterruptedException {
10         Thread a = new Thread(() -> task(), "A");
11         Thread b = new Thread(() -> task(), "B");
▶          a.start();
13         b.start();
14         a.join();
15         b.join();
16         System.out.println("main: done");
17     }
18 }
```



main starts A. A is now `RUNNABLE`; when it will run is unknown.

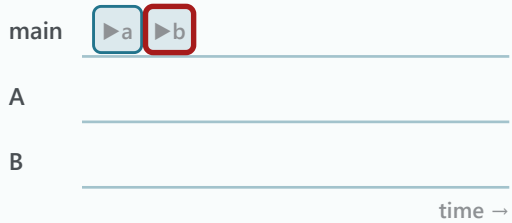
Step 1 / 8

OUTPUT

One of the possible orders. A real run gave `B1 · B2 · A1 · A2`. What is certain: “main: done” comes last.

Two Threads: One Possible Order

```
1  public class TwoThreads {
2      static void task() {
3          String name = Thread.currentThread().getName();
4          for (int i = 1; i <= 2; i++) {
5              System.out.println(name + ": step " + i);
6          }
7      }
8      public static void main(String[] args)
9              throws InterruptedException {
10         Thread a = new Thread(() -> task(), "A");
11         Thread b = new Thread(() -> task(), "B");
12         a.start();
▶          b.start();
14         a.join();
15         b.join();
16         System.out.println("main: done");
17     }
18 }
```



main starts B.

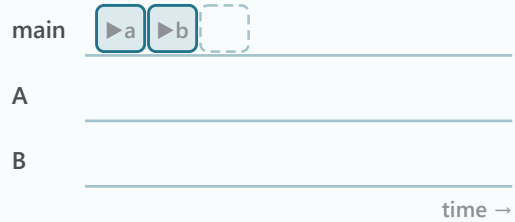
Step 2 / 8

OUTPUT

One of the possible orders. A real run gave `B1 · B2 · A1 · A2`. What is certain: “main: done” comes last.

Two Threads: One Possible Order

```
1  public class TwoThreads {
2      static void task() {
3          String name = Thread.currentThread().getName();
4          for (int i = 1; i <= 2; i++) {
5              System.out.println(name + ": step " + i);
6          }
7      }
8      public static void main(String[] args)
9              throws InterruptedException {
10         Thread a = new Thread(() -> task(), "A");
11         Thread b = new Thread(() -> task(), "B");
12         a.start();
13         b.start();
▶          a.join();
15         b.join();
16         System.out.println("main: done");
17     }
18 }
```



`a.join()` : main is `WAITING` until A ends. Now A and B run.

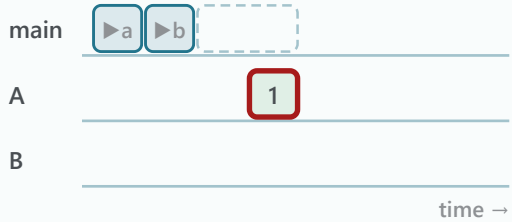
Step 3 / 8

OUTPUT

One of the possible orders. A real run gave `B1 · B2 · A1 · A2`. What is certain: “main: done” comes last.

Two Threads: One Possible Order

```
1  public class TwoThreads {
2      static void task() {
3          String name = Thread.currentThread().getName();
4          for (int i = 1; i <= 2; i++) {
5              System.out.println(name + ": step " + i);
6          }
7      }
8      public static void main(String[] args)
9              throws InterruptedException {
10         Thread a = new Thread(() -> task(), "A");
11         Thread b = new Thread(() -> task(), "B");
12         a.start();
13         b.start();
14         a.join();
15         b.join();
16         System.out.println("main: done");
17     }
18 }
```



The scheduler runs A: A executes `task()` on its own stack.

Step 4 / 8

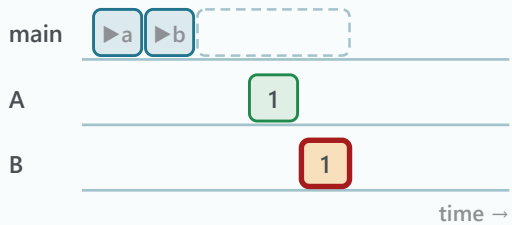
OUTPUT

```
A: step 1
```

One of the possible orders. A real run gave `B1 · B2 · A1 · A2`. What is certain: “main: done” comes last.

Two Threads: One Possible Order

```
1  public class TwoThreads {
2      static void task() {
3          String name = Thread.currentThread().getName();
4          for (int i = 1; i <= 2; i++) {
▶              System.out.println(name + ": step " + i);
6          }
7      }
8      public static void main(String[] args)
9              throws InterruptedException {
10         Thread a = new Thread(() -> task(), "A");
11         Thread b = new Thread(() -> task(), "B");
12         a.start();
13         b.start();
14         a.join();
15         b.join();
16         System.out.println("main: done");
17     }
18 }
```



A context switch: now B is running. B's `i` is a separate local variable.

Step 5 / 8

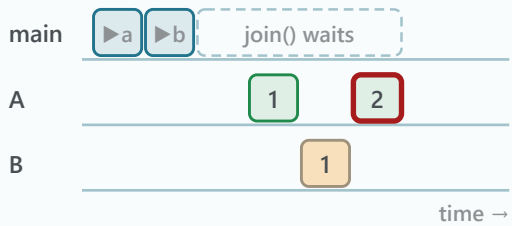
OUTPUT

```
A: step 1
B: step 1
```

One of the possible orders. A real run gave `B1 · B2 · A1 · A2`. What is certain: “main: done” comes last.

Two Threads: One Possible Order

```
1  public class TwoThreads {
2      static void task() {
3          String name = Thread.currentThread().getName();
4          for (int i = 1; i <= 2; i++) {
5              System.out.println(name + ": step " + i);
6          }
7      }
8      public static void main(String[] args)
9              throws InterruptedException {
10         Thread a = new Thread(() -> task(), "A");
11         Thread b = new Thread(() -> task(), "B");
12         a.start();
13         b.start();
14         a.join();
15         b.join();
16         System.out.println("main: done");
17     }
18 }
```



A again.

Step 6 / 8

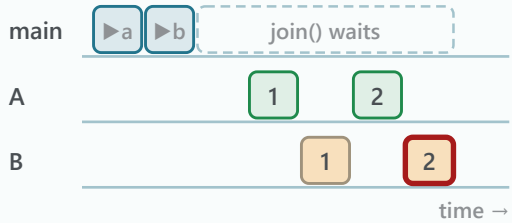
OUTPUT

```
A: step 1
B: step 1
A: step 2
```

One of the possible orders. A real run gave `B1 · B2 · A1 · A2`. What is certain: “main: done” comes last.

Two Threads: One Possible Order

```
1  public class TwoThreads {
2      static void task() {
3          String name = Thread.currentThread().getName();
4          for (int i = 1; i <= 2; i++) {
▶              System.out.println(name + ": step " + i);
6          }
7      }
8      public static void main(String[] args)
9              throws InterruptedException {
10         Thread a = new Thread(() -> task(), "A");
11         Thread b = new Thread(() -> task(), "B");
12         a.start();
13         b.start();
14         a.join();
15         b.join();
16         System.out.println("main: done");
17     }
18 }
```



B again. Then both end (`TERMINATED`).

Step 7 / 8

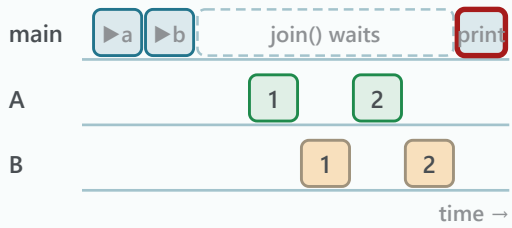
OUTPUT

```
A: step 1
B: step 1
A: step 2
B: step 2
```

One of the possible orders. A real run gave `B1 · B2 · A1 · A2` . What is certain: “main: done” comes last.

Two Threads: One Possible Order

```
1  public class TwoThreads {
2      static void task() {
3          String name = Thread.currentThread().getName();
4          for (int i = 1; i <= 2; i++) {
5              System.out.println(name + ": step " + i);
6          }
7      }
8      public static void main(String[] args)
9              throws InterruptedException {
10         Thread a = new Thread(() -> task(), "A");
11         Thread b = new Thread(() -> task(), "B");
12         a.start();
13         b.start();
14         a.join();
15         b.join();
▶          System.out.println("main: done");
17     }
18 }
```



`a.join()` returns; `b.join()` returns at once since B is already done; main prints.

Step 8 / 8

OUTPUT

```
A: step 1
B: step 1
A: step 2
B: step 2
main: done
```

One of the possible orders. A real run gave `B1 · B2 · A1 · A2`. What is certain: “main: done” comes last.

Managing Threads: sleep, join, yield

sleep(ms)
Puts the current thread to **sleep** for a while (`TIMED_WAITING`). It keeps any lock it holds.

join()
The calling thread **waits until** the target thread ends.
`join(ms)` : at most ms.

yield()
A **hint** to let another thread run for now; the scheduler may ignore it.

```
public class JoinDemo {
    static long result = 0;
    public static void main(String[] args)
            throws InterruptedException {
        Thread calc = new Thread(() -> {
            long sum = 0;
            for (int i = 1; i <= 1_000_000; i++) {
                sum += i;
            }
            result = sum;
        });
        calc.start();
        System.out.println("before join: " + result);
        calc.join();   // wait for calc to end
        System.out.println("after join : " + result);
    }
}
```

Output

```
before join: 0
after join : 500000500000
```

Without `join()` , main reads `result` before calc finishes and sees **0**. After `join()` , calc has definitely finished and the value it wrote is visible.

`sleep` and `join` can throw `InterruptedException` : catch it or declare it with `throws` .

Daemon (Background) Threads

```
public class DaemonDemo {
    public static void main(String[] args)
            throws InterruptedException {
        Thread clock = new Thread(() -> {
            int s = 0;
            while (true) {
                System.out.println("tick " + s++);
                try {
                    Thread.sleep(300);
                } catch (InterruptedException e) {
                    return;
                }
            }
        });
        clock.setDaemon(true);   // BEFORE start()
        clock.start();
        Thread.sleep(1000);
        System.out.println("main done, JVM exiting");
    }
}
```

Output

```
tick 0
tick 1
tick 2
tick 3
main done, JVM exiting
```

- The JVM exits once all **non-daemon** threads have ended; it doesn't wait for daemon threads.
- Though `clock` loops forever, the program ends when main does.
- Call `setDaemon(true)` **before** `start()`; afterwards it throws `IllegalThreadStateException`.

Stopping a Thread

- A thread stops when its `run()` method completes.
- The JVM exits when `System.exit()` (internally `Runtime.exit()`) is called or when all non-daemon threads have ended.
- The right way to stop a thread from outside is **cooperation**: the thread periodically checks a **flag**.
- The flag is declared `volatile` so other threads can see the change.
- Alternative: an interrupt request with `interrupt()`.

Don't use Thread.stop()

The deprecated `stop()` kills a thread mid-task; its locks are released and objects may be left half-updated. Since Java 20 it no longer works:

```
Thread t = new Thread(() -> { });
try {
    t.stop();
} catch (UnsupportedOperationException e) {
    System.out.println("stop(): not supported");
}
```

Output (JDK 21)

```
stop(): not supported
```

Stopping with a Flag

```
class Worker extends Thread {
    private volatile boolean flag = false;
    public void finish() { flag = true; }
    public void run() {
        while (!flag) {
            System.out.println("working...");
            try { Thread.sleep(100); }
            catch (InterruptedException e) { return; }
        }
        System.out.println("flag seen, finishing");
    }
}
public class FlagStop {
    public static void main(String[] args)
            throws InterruptedException {
        Worker w = new Worker();
        w.start();
        Thread.sleep(350);
        w.finish();   // raise the flag
        w.join();     // wait for it to end
        System.out.println("main: worker stopped");
    }
}
```

Output

```
working...
working...
working...
working...
flag seen, finishing
main: worker stopped
```

- `finish()` only raises the flag; the thread sees it at the top of the loop and leaves `run()` **on its own**.
- While main slept 350 ms, the worker printed four times, at 0, 100, 200 and 300 ms.
- Without `volatile`, the worker might never see the flag change.

Stopping via a Reference

```
class Worker implements Runnable {
    private volatile Thread thread;
    public void start() {
        thread = new Thread(this);
        thread.start();
    }
    public void finish() { thread = null; }
    public void run() {
        Thread me = Thread.currentThread();
        while (thread == me) {   // ... work ...
        }
        System.out.println("run() done");
    }
}
public class ReferenceStop {
    public static void main(String[] args) throws Exception {
        Worker w = new Worker();
        w.start();
        Thread.sleep(200);
        w.finish();
        Thread.sleep(200);
        System.out.println("main: done");
    }
}
```

Output

```
run() done
main: done
```

- The class implements `Runnable`; the running `Thread` object is kept in a `volatile` field.
- The loop goes on while the field still points to **its own** thread, compared with `Thread.currentThread()`.
- `finish()` sets the field to `null`; the condition fails and `run()` ends.
- `currentThread()` is *static*, so a class that doesn't extend `Thread` needs the `Thread.` prefix.

Executing Operations in the Right Order

Unknown speed

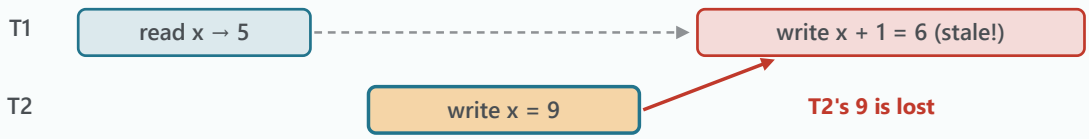
Each thread's speed and frequency vary. Operating systems may **ignore** priorities; you can't rely on priorities for ordering.

Stale values

After one thread reads a variable, another may update it; the first carries on with an **out-of-date** value.

Reordering

The compiler, the JIT and the CPU may reorder instructions for speed; unnoticeable in one thread, surprising with several.



When several threads access shared, **mutable** data, access must be **synchronized**. Otherwise the result depends on the order in which the threads run: this is a **race condition**.

Race Conditions

```
public class RaceCondition {
    static int counter = 0;   // shared data
    public static void main(String[] args)
            throws InterruptedException {
        Runnable inc = () -> {
            for (int i = 0; i < 100_000; i++) {
                counter++;   // read, add 1, write: NOT atomic
            }
        };
        Thread t1 = new Thread(inc);
        Thread t2 = new Thread(inc);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Expected: 200000");
        System.out.println("Actual  : " + counter);
    }
}
```

Run 1

```
Expected: 200000
Actual  : 138001
```

Run 2

```
Expected: 200000
Actual  : 135352
```

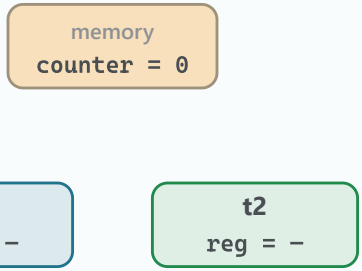
Run 3

```
Expected: 200000
Actual  : 136776
```

A **different** result below 200000 every run: some increments get lost.

Why Does counter++ Get Lost?

```
// counter++ is three CPU steps:
int reg = counter;      // 1) read
reg = reg + 1;          // 2) add 1
counter = reg;          // 3) write
```



At first `counter = 0`. Both threads will do `counter++`; the right result is 2.

Step 1 / 7

VARIABLES

`counter = 0` `t1.reg = –` `t2.reg = –`

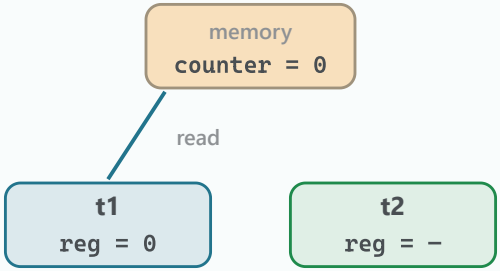
CALL STACK

empty

This is just one possible order; without an interleaving the result is right. That is why the bug looks **random** and is hard to test. The call stack box shows the thread running at that moment.

Why Does counter++ Get Lost?

```
1 // counter++ is three CPU steps:
▶ int reg = counter;      // 1) read
3 reg = reg + 1;          // 2) add 1
4 counter = reg;          // 3) write
```



t1 reads: its register gets 0.

Step 2 / 7

VARIABLES

`counter = 0` `t1.reg = 0` `t2.reg = –`

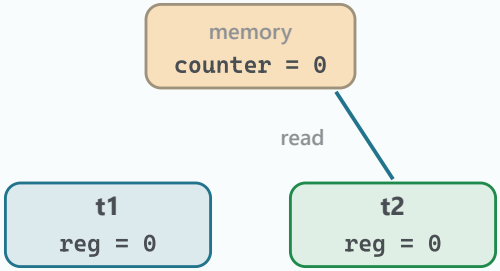
CALL STACK

`t1`

This is just one possible order; without an interleaving the result is right. That is why the bug looks **random** and is hard to test. The call stack box shows the thread running at that moment.

Why Does counter++ Get Lost?

```
1 // counter++ is three CPU steps:
▶ int reg = counter;      // 1) read
3 reg = reg + 1;          // 2) add 1
4 counter = reg;          // 3) write
```



Context switch! t2 reads too: it also sees 0.

Step 3 / 7

VARIABLES

`counter = 0` `t1.reg = 0` `t2.reg = 0`

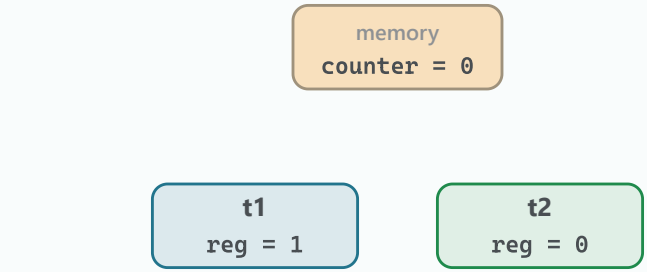
CALL STACK

`t2`

This is just one possible order; without an interleaving the result is right. That is why the bug looks **random** and is hard to test. The call stack box shows the thread running at that moment.

Why Does counter++ Get Lost?

```
1 // counter++ is three CPU steps:
2 int reg = counter;      // 1) read
▶ reg = reg + 1;          // 2) add 1
4 counter = reg;          // 3) write
```



t1 adds 1 in its register.

Step 4 / 7

VARIABLES

`counter = 0` `t1.reg = 1` `t2.reg = 0`

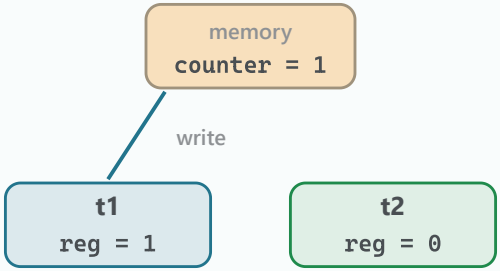
CALL STACK

`t1`

This is just one possible order; without an interleaving the result is right. That is why the bug looks **random** and is hard to test. The call stack box shows the thread running at that moment.

Why Does counter++ Get Lost?

```
1 // counter++ is three CPU steps:
2 int reg = counter;      // 1) read
3 reg = reg + 1;          // 2) add 1
▶ counter = reg;          // 3) write
```



t1 writes: `counter = 1` .

Step 5 / 7

VARIABLES

`counter = 1` `t1.reg = 1` `t2.reg = 0`

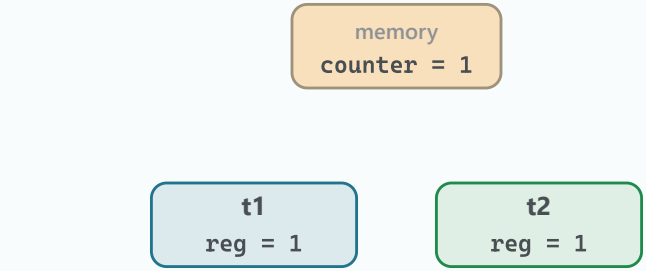
CALL STACK

`t1`

This is just one possible order; without an interleaving the result is right. That is why the bug looks **random** and is hard to test. The call stack box shows the thread running at that moment.

Why Does counter++ Get Lost?

```
1 // counter++ is three CPU steps:
2 int reg = counter;      // 1) read
▶ reg = reg + 1;          // 2) add 1
4 counter = reg;          // 3) write
```



t2 adds 1 to its own (stale) value: 1.

Step 6 / 7

VARIABLES

`counter = 1` `t1.reg = 1` `t2.reg = 1`

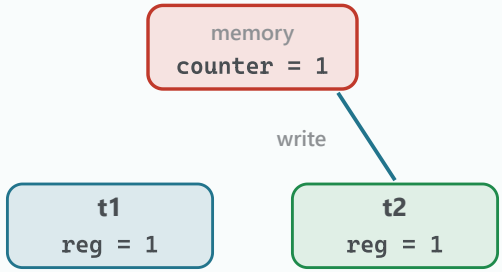
CALL STACK

`t2`

This is just one possible order; without an interleaving the result is right. That is why the bug looks **random** and is hard to test. The call stack box shows the thread running at that moment.

Why Does counter++ Get Lost?

```
1 // counter++ is three CPU steps:
2 int reg = counter;      // 1) read
3 reg = reg + 1;          // 2) add 1
▶ counter = reg;          // 3) write
```



t2 writes: `counter = 1`. Two increments ran, but the result is 1: t1's increment is **lost**.

Step 7 / 7

VARIABLES

`counter = 1` `t1.reg = 1` `t2.reg = 1`

CALL STACK

`t2`

This is just one possible order; without an interleaving the result is right. That is why the bug looks **random** and is hard to test. The call stack box shows the thread running at that moment.

Synchronization Options

join()
Wait for a thread to finish.

synchronized
Mutual exclusion with an object's monitor lock.

Lock objects
`ReentrantLock`, `ReadWriteLock`: explicit locking.

Barriers, semaphores, exchangers, latches
`CyclicBarrier`, `Semaphore`, `Exchanger`, `CountDownLatch`.

volatile variables
Reads and writes visible to all threads.

Atomic objects
Indivisible operations such as `AtomicInteger`.

Things that look like synchronization but aren't: priorities, sleeping (`sleep`), yielding (`yield`) and timers. They **don't guarantee** order; never write code that says “I'll wait a bit, the other one will be done by then”.

synchronized and Monitors

- Every object has a **monitor** (intrinsic lock) that can be locked and unlocked.
- Only **one** thread can own a monitor at a time.
- If t2 wants it while t1 owns it, t2 waits (`BLOCKED`).
- When the monitor is released, the waiting threads **compete** and only one gets it.
- The monitor is taken with `synchronized` and released automatically when the block ends (even via an exception).
- This gives **mutual exclusion** and protects data integrity.

SYNTAX

```
synchronized (o) {
    // we hold o's monitor here
}
```

```
Runnable r = () -> {
    for (int i = 0; i < 50_000; i++) {
        synchronized (o) {   // take o's monitor
            shared++;
        }                    // monitor released
    }
};
```

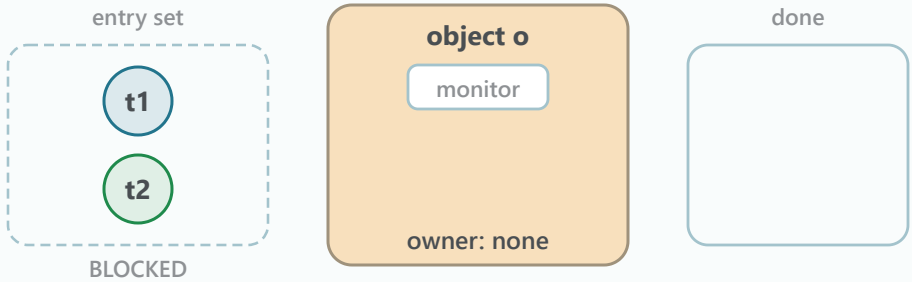
Output (2 × 50000, every time)

```
100000
```

Monitors Step by Step

```
// t1 and t2 run the same code
synchronized (o) {
    shared++;     // critical section
}
```

A **critical section** is code that accesses shared data and must be run by only one thread at a time.

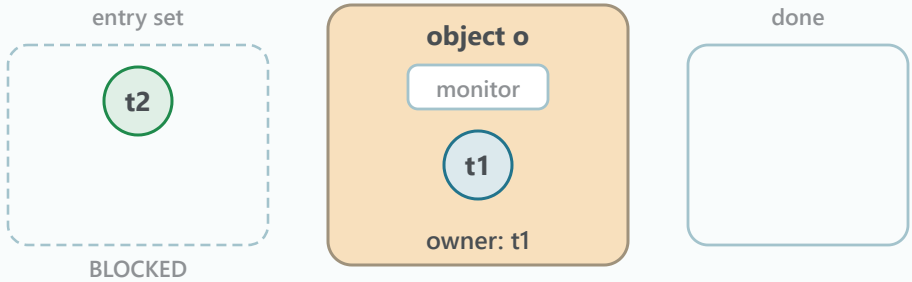


t1 and t2 reach the `synchronized (o)` line at the same time; the monitor is free.

Monitors Step by Step

```
// t1 and t2 run the same code
synchronized (o) {
    shared++;     // critical section
}
```

A **critical section** is code that accesses shared data and must be run by only one thread at a time.

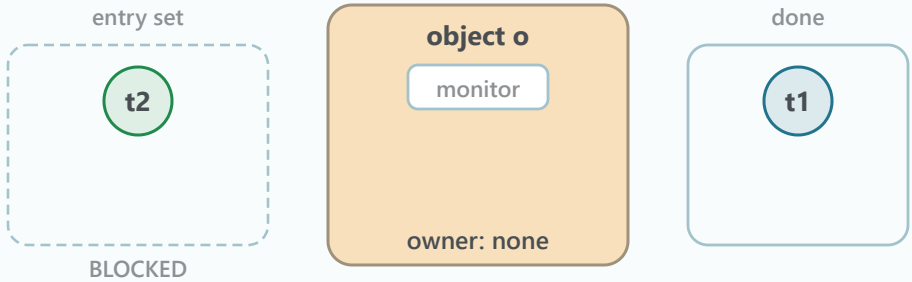


t1 wins and takes the monitor. t2 waits in the entry set, `BLOCKED`.

Monitors Step by Step

```
// t1 and t2 run the same code
synchronized (o) {
    shared++;     // critical section
}
```

A **critical section** is code that accesses shared data and must be run by only one thread at a time.

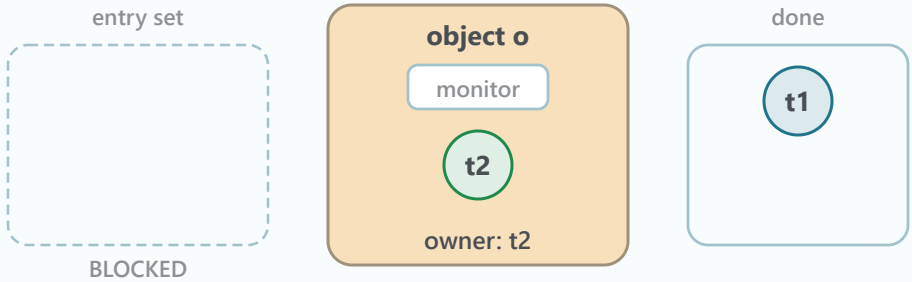


t1 leaves the block: the monitor is released automatically. The waiting threads compete.

Monitors Step by Step

```
// t1 and t2 run the same code
synchronized (o) {
    shared++;     // critical section
}
```

A **critical section** is code that accesses shared data and must be run by only one thread at a time.

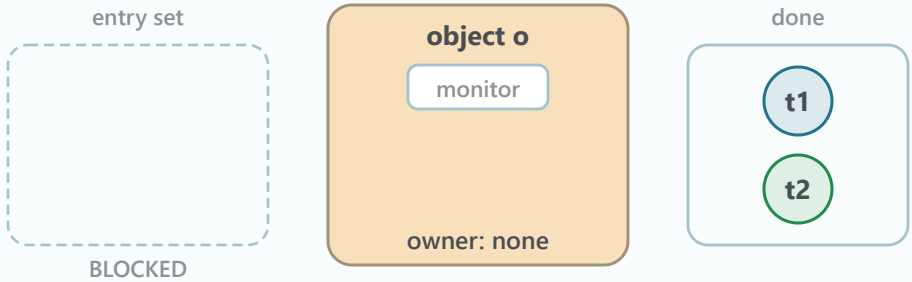


t2 takes the monitor and enters the critical section.

Monitors Step by Step

```
// t1 and t2 run the same code
synchronized (o) {
    shared++;     // critical section
}
```

A **critical section** is code that accesses shared data and must be run by only one thread at a time.



t2 leaves too. `shared++` ran twice without interfering.

Declaring Synchronized Methods

`synchronized` METHOD

```
class C {
    synchronized void p() { ... }
    static synchronized void q() { ... }
    //...
}
```

EQUIVALENT: A `synchronized` BLOCK

```
class C {
    void p() { synchronized (this) { ... } }
    static void q() { synchronized (C.class) { ... } }
    //...
}
```

- Declaring a method `synchronized` is **the same** as wrapping its body in a `synchronized` block.
- Instance method → the monitor of `this`.
- `static` method → the monitor of the class object `C.class` (there is no instance).
- So a `static synchronized` method and an instance `synchronized` method **don't block** each other: different locks.
- The block form lets you lock only the lines that really need it.

With real bodies in place of `...` (say `n++`, `s++`), both forms compile and behave the same.

Preventing the Race

```
class Counter {
    private int value = 0;
    public synchronized void increment() { value++; }
    public synchronized int getValue() { return value; }
}

public class SyncCounter {
    public static void main(String[] args)
            throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 100_000; i++) {
                counter.increment();
            }
        };
        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Result: " + counter.getValue());
    }
}
```

Output (every run)

```
Result: 200000
```

- `increment()` is now `synchronized`: the read–add–write steps run without interruption.
- The reader, `getValue()`, must be synchronized too; otherwise it isn't guaranteed to see the latest value.
- The price: threads enter one at a time and slow down a bit. So protect only shared data.

Mutual Exclusion: MinQueue

```
class MinQueue {
    private int[] data;
    private int size;
    public MinQueue(int n) { data = new int[n]; }
    public synchronized void add(int o) {
        if (size == data.length)
            throw new HeapFullException();
        data[size++] = o;
    }
    public synchronized int remove() {   // the smallest
        int min = 0;
        for (int i = 1; i < size; i++) {
            if (data[i] < data[min]) min = i;
        }
        int result = data[min];
        data[min] = data[--size];
        return result;
    }
    public synchronized String toString() {
        return Arrays.toString(Arrays.copyOf(data, size));
    }
}
```

All methods use the same monitor (`this`).

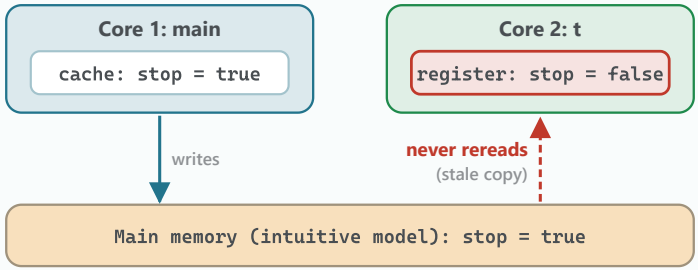
```
MinQueue q =
        new MinQueue(3);
q.add(5);
q.add(2);
q.add(8);
System.out.println(q);
int x = q.remove();
System.out.println(x);
System.out.println(q);
q.add(1);
try {
    q.add(9);
} catch (HeapFullException e) {
    String m = e.getMessage();
    System.out.println(m);
}
```

Output

```
[5, 2, 8]
2
[5, 8]
Queue full
```

volatile Fields

- For speed, values may be kept in a cache or a **register**; other threads then can't see the current value.
- For a `volatile` field, **every read sees the last write** to it (a volatile write *happens-before* the next read).
- Single reads and writes of variables other than `long` and `double` are already atomic; `volatile long/double` become atomic too.
- A simple shared flag needs no `synchronized` or lock.
- But only reads and writes are atomic: read–modify–write like `x++` is **not atomic**.



The Visibility Problem

```
public class VisibilityProblem {
    static boolean stop = false;   // NOT volatile!
    public static void main(String[] args)
            throws InterruptedException {
        Thread t = new Thread(() -> {
            long laps = 0;
            while (!stop) {
                laps++;
            }
            System.out.println("thread stopped");
        });
        t.setDaemon(true);
        t.start();
        Thread.sleep(500);
        stop = true;
        System.out.println("main: stop = true");
        t.join(2000);   // wait at most 2 s
        boolean alive = t.isAlive();
        System.out.println("still running? " + alive);
    }
}
```

Output: NO volatile

```
main: stop = true
still running? true
```

Output: static volatile boolean stop

```
main: stop = true
thread stopped
still running? false
```

Without `volatile`, the JIT compiler may read `stop` once and run the loop as `while (true)`: the thread **never sees** the change. Results vary by JVM; correctness shouldn't be left to luck.

`setDaemon(true)`: so the stuck thread doesn't keep the program from exiting.

volatile Is Not Atomic and Can't Be final

```
static volatile int counter = 0;
// inside main:
Runnable r = () -> {
    for (int i = 0; i < 100_000; i++) counter++;
};
```

Three runs (expected 200000)

```
volatile counter: 163299
volatile counter: 164441
volatile counter: 199332
```

Visibility is fine, but two threads can still read the same value and write the same result. The fix: `synchronized` or `AtomicInteger`.

```
public class FinalVolatile {
    private final volatile int x = 5;
}
```

Compiler error

```
FinalVolatile.java:2: error: illegal
combination of modifiers: final and
volatile
    private final volatile int x = 5;
                               ^
1 error
```

final	volatile
Assigned once (in the constructor); never changes	Its value can change
After the constructor (if `this` didn't escape), all see the same value	Every read returns the current value

They solve synchronization needs differently; they can't be combined.

VISIBILITY

The Java Memory Model: happens-before

The Java Memory Model (JMM) defines **when** a value written by one thread becomes visible to another. Programmers reason not with CPU barriers but with **happens-before** relations: if A happens-before B, whatever A wrote is guaranteed to be visible in B.

Rule	A happens-before B
Locks	**releasing** a monitor/lock → the next **acquisition** of the same lock
`volatile`	a **write** to a `volatile` field → the next **read** of that field
Start	everything before `t.start()` → the actions inside `t`
Termination	all of `t`'s actions → the thread returning from `t.join()`
Library structures	`BlockingQueue.put` → the `take` that gets that element; a `Future`'s task → `get()`

False sharing: if **independent** variables of different threads sit on the same cache line (usually 64 bytes) and are updated often, the cores keep pulling the line from each other. The result isn't wrong, but the program **slows down**: a performance problem, not a correctness one.

Explicit Locking

```
public class LockDemo {
    private final Lock lock = new ReentrantLock();
    private int counter = 0;
    public void work() {
        lock.lock();
        try {
            // critical section
            counter++;
        } finally {
            lock.unlock();
        }
    }
    public static void main(String[] args)
            throws InterruptedException {
        LockDemo o = new LockDemo();
        Runnable r = () -> {
            for (int i = 0; i < 100_000; i++) o.work();
        };
        Thread a = new Thread(r), b = new Thread(r);
        a.start(); b.start();
        a.join(); b.join();
        System.out.println("Result: " + o.counter);
    }
}
```

- Used for complex synchronization scenarios.
- The `Lock` interface in `java.util.concurrent.locks`; `ReentrantLock` and `ReadWriteLock` offer advanced, customizable locks.
- Adds flexibility and performance: try-locks, timed and interruptible lock acquisition.
- Use with care: **you** must release the lock.
- Import: `java.util.concurrent.locks.*`

Output

```
Result: 200000
```

Unlock in a finally Block

THE RIGHT PATTERN

```
public void work() {
    Lock lock = ...;
    lock.lock();
    try {
        // access the guarded resource
    } finally {
        lock.unlock();
    }
}
```

- Call `lock()` right **before** the `try`.
- `unlock()` goes in `finally`: the lock is released **in every case**, even if the critical section throws.

WRONG: NO `finally`

```
lock.lock();
doWork();       // if this throws...
lock.unlock();  // ...we never get here
```

The lock stays locked; every thread that wants it waits forever. `synchronized` has no such problem: the lock is released automatically when the block ends.

WRONG: UNLOCKING A LOCK YOU DON'T HOLD

```
Lock lock = new ReentrantLock();
lock.unlock();   // never acquired
```

Runtime error

```
Exception in thread "main"
java.lang.IllegalMonitorStateException
```

tryLock(): Locking Without Waiting, or With a Timeout

```
static void work()
        throws InterruptedException {
    if (lock.tryLock()) {
        // got it
        try { System.out.println("1: acquired"); }
        finally { lock.unlock(); }
    } else {
        // didn't get it; do something else
        System.out.println("1: not acquired");
    }
    if (lock.tryLock(3, TimeUnit.SECONDS)) {
        try { System.out.println("2: acquired"); }
        finally { lock.unlock(); }
    } else {
        // not acquired within 3 seconds
        System.out.println("2: not acquired");
    }
}
```

- `tryLock()` : takes the lock if free and returns `true` ; if not, returns `false` **without waiting**.
- `tryLock(3, SECONDS)` : waits at most 3 s; `InterruptedException` must be declared.

```
lock.lock();   // main holds the lock
Thread t = new Thread(() -> {
    try { work(); }
    catch (InterruptedException e) { }
});
t.start();
Thread.sleep(1000);
System.out.println("main: releasing the lock");
lock.unlock();
t.join();
```

Output

```
1: not acquired
main: releasing the lock
2: acquired
```

The ReentrantLock Class

- The name means it allows **reentry**.
- The thread holding the lock can take **the same lock again**; it doesn't wait, a counter just goes up. The lock is released when the counter drops to 0.
- Several `ReentrantLock`s can guard nested critical sections.
- **Fair locking**: with `new ReentrantLock(true)` the lock goes to the longest-waiting thread.
- `synchronized` is reentrant too: a `synchronized` method can call another `synchronized` method of the same object.

```
ReentrantLock lock1 = new ReentrantLock();
// true: a fair lock
ReentrantLock lock2 = new ReentrantLock(true);
// ...
lock1.lock();
try {
    // enter the critical section
    lock2.lock();
    try {
        // enter the nested section
        System.out.println("both locks acquired");
    } finally {  lock2.unlock();  }
} finally {  lock1.unlock();  }
System.out.println("fair? " + lock2.isFair());
```

Output

```
both locks acquired
fair? true
```

Reentrancy, Step by Step

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  public class Reentrancy {
4      static final ReentrantLock lock =
5              new ReentrantLock();
6      static void show(String where) {
7          System.out.println(where + lock.getHoldCount());
8      }
9      static void outer() {
10         lock.lock();
11         try { show("outer(): "); inner(); }
12         finally { lock.unlock(); }
13     }
14     static void inner() {
15         lock.lock();   // enters without waiting
16         try { show("inner(): "); }
17         finally { lock.unlock(); }
18     }
19     public static void main(String[] args) {
▶            outer();
21         show("main:    ");
22     }
23 }
```

lock

owner: none



count = 0

main calls `outer()`. The lock is free.

Step 1 / 8

CALL STACK

`main()`

OUTPUT

Reentrancy, Step by Step

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  public class Reentrancy {
4      static final ReentrantLock lock =
5              new ReentrantLock();
6      static void show(String where) {
7          System.out.println(where + lock.getHoldCount());
8      }
9      static void outer() {
▶          lock.lock();
11         try { show("outer(): "); inner(); }
12         finally { lock.unlock(); }
13     }
14     static void inner() {
15         lock.lock();   // enters without waiting
16         try { show("inner(): "); }
17         finally { lock.unlock(); }
18     }
19     public static void main(String[] args) {
20         outer();
21         show("main:    ");
22     }
23 }
```



`outer()` takes the lock: owner main, hold count 1.

Step 2 / 8

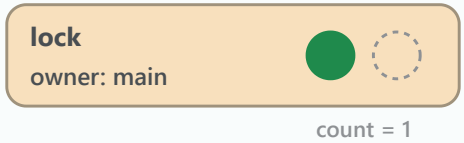
CALL STACK

- `outer()`
- `main()`

OUTPUT

Reentrancy, Step by Step

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  public class Reentrancy {
4      static final ReentrantLock lock =
5              new ReentrantLock();
6      static void show(String where) {
7          System.out.println(where + lock.getHoldCount());
8      }
9      static void outer() {
10         lock.lock();
▶          try { show("outer(): "); inner(); }
12         finally { lock.unlock(); }
13     }
14     static void inner() {
15         lock.lock();   // enters without waiting
16         try { show("inner(): "); }
17         finally { lock.unlock(); }
18     }
19     public static void main(String[] args) {
20         outer();
21         show("main:    ");
22     }
23 }
```



`show` prints the count.

Step 3 / 8

CALL STACK

- `show()`
- `outer()`
- `main()`

OUTPUT

```
outer(): 1
```

Reentrancy, Step by Step

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  public class Reentrancy {
4      static final ReentrantLock lock =
5              new ReentrantLock();
6      static void show(String where) {
7          System.out.println(where + lock.getHoldCount());
8      }
9      static void outer() {
10         lock.lock();
11         try { show("outer(): "); inner(); }
12         finally { lock.unlock(); }
13     }
14     static void inner() {
▶          lock.lock();   // enters without waiting
16         try { show("inner(): "); }
17         finally { lock.unlock(); }
18     }
19     public static void main(String[] args) {
20         outer();
21         show("main:    ");
22     }
23 }
```



`inner()` asks for the same lock. main already owns it, so it **doesn't wait**: the count becomes 2.

Step 4 / 8

CALL STACK

- `inner()`
- `outer()`
- `main()`

OUTPUT

```
outer(): 1
```

Reentrancy, Step by Step

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  public class Reentrancy {
4      static final ReentrantLock lock =
5              new ReentrantLock();
6      static void show(String where) {
7          System.out.println(where + lock.getHoldCount());
8      }
9      static void outer() {
10         lock.lock();
11         try { show("outer(): "); inner(); }
12         finally { lock.unlock(); }
13     }
14     static void inner() {
15         lock.lock();   // enters without waiting
▶          try { show("inner(): "); }
17         finally { lock.unlock(); }
18     }
19     public static void main(String[] args) {
20         outer();
21         show("main:    ");
22     }
23 }
```



Printed.

Step 5 / 8

CALL STACK

- **show()**
- inner()
- outer()
- main()

OUTPUT

```
outer(): 1
inner(): 2
```

Reentrancy, Step by Step

```
 1 import java.util.concurrent.locks.ReentrantLock;
 2
 3 public class Reentrancy {
 4     static final ReentrantLock lock =
 5             new ReentrantLock();
 6     static void show(String where) {
 7         System.out.println(where + lock.getHoldCount());
 8     }
 9     static void outer() {
10         lock.lock();
11         try { show("outer(): "); inner(); }
12         finally { lock.unlock(); }
13     }
14     static void inner() {
15         lock.lock();   // enters without waiting
16         try { show("inner(): "); }
▶           finally { lock.unlock(); }
18     }
19     public static void main(String[] args) {
20         outer();
21         show("main:    ");
22     }
23 }
```



`inner()` 's `finally` : `unlock()` drops the count to 1; main still holds the lock.

Step 6 / 8

CALL STACK

- `inner()`
- `outer()`
- `main()`

OUTPUT

```
outer(): 1
inner(): 2
```

Reentrancy, Step by Step

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  public class Reentrancy {
4      static final ReentrantLock lock =
5              new ReentrantLock();
6      static void show(String where) {
7          System.out.println(where + lock.getHoldCount());
8      }
9      static void outer() {
10         lock.lock();
11         try { show("outer(): "); inner(); }
▶          finally { lock.unlock(); }
13     }
14     static void inner() {
15         lock.lock();   // enters without waiting
16         try { show("inner(): "); }
17         finally { lock.unlock(); }
18     }
19     public static void main(String[] args) {
20         outer();
21         show("main:    ");
22     }
23 }
```

lock

owner: none



count = 0

`outer()` 's `finally` : the count is 0 and the lock is really released.

Step 7 / 8

CALL STACK

- `outer()`
- `main()`

OUTPUT

```
outer(): 1
inner(): 2
```

Reentrancy, Step by Step

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  public class Reentrancy {
4      static final ReentrantLock lock =
5              new ReentrantLock();
6      static void show(String where) {
7          System.out.println(where + lock.getHoldCount());
8      }
9      static void outer() {
10         lock.lock();
11         try { show("outer(): "); inner(); }
12         finally { lock.unlock(); }
13     }
14     static void inner() {
15         lock.lock();   // enters without waiting
16         try { show("inner(): "); }
17         finally { lock.unlock(); }
18     }
19     public static void main(String[] args) {
20         outer();
▶            show("main:    ");
22     }
23 }
```

lock
owner: none



count = 0

Every `lock()` needs one `unlock()`.

Step 8 / 8

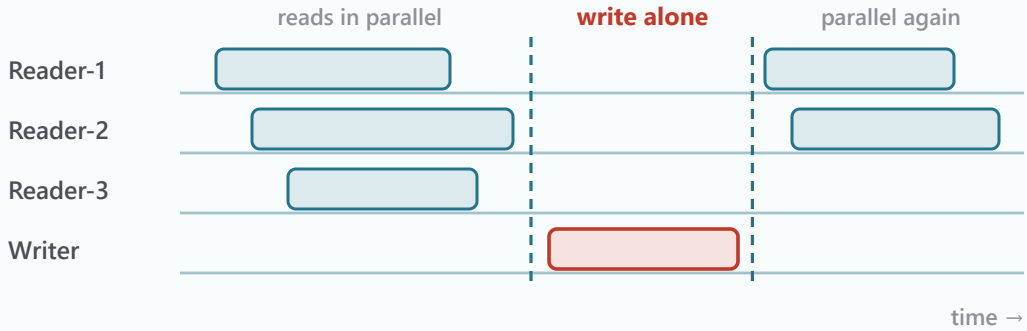
CALL STACK

- **show()**
- main()

OUTPUT

```
outer(): 1
inner(): 2
main:    0
```

The ReadWriteLock Class



Read lock
Several threads can read the data **at the same time** without blocking each other (parallel reads).

Write lock
While one thread writes, others can neither read nor write; the write is safe and atomic.

When?
Improves performance when data is **read often and written rarely** (caches, settings tables).

ReadWriteLock Code

```
private final Map<String, String> table =
        new HashMap<>();
private final ReadWriteLock lock =
        new ReentrantReadWriteLock();

public String get(String key) {
    lock.readLock().lock();   // lock for reading
    try {
        return table.get(key);   // read the data
    } finally {
        lock.readLock().unlock();
    }
}
public void put(String key, String value) {
    lock.writeLock().lock();  // lock for writing
    try {
        table.put(key, value);   // write the data
    } finally {
        lock.writeLock().unlock();
    }
}
```

```
Settings s = new Settings();
s.put("lang", "en");
Runnable reader = () -> System.out.println(
        Thread.currentThread().getName()
        + ": " + s.get("lang"));
Thread r1 = new Thread(reader, "Reader-1");
Thread r2 = new Thread(reader, "Reader-2");
r1.start(); r2.start();
r1.join(); r2.join();
```

Output (one run)

```
Reader-2: en
Reader-1: en
```

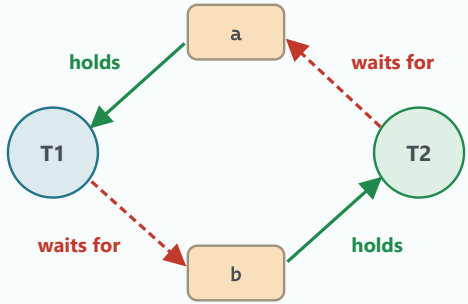
Two readers can hold the read lock at once; their order may vary. `ReentrantReadWriteLock` implements the `ReadWriteLock` interface.

Explicit vs Implicit Locking

	Explicit: `Lock`	**Implicit:** `synchronized`
Control	The developer calls `lock()` / `unlock()`	Automatic: taken on block entry, released on exit
Classes	The `Lock` interface, `ReentrantLock`, `ReentrantReadWriteLock`	A keyword; every object's monitor
Try without waiting	`tryLock()`, `tryLock(time)`: conditional, non-blocking	None: waits until the lock is free
Interruptible wait	`lockInterruptibly()`	None
Fairness, conditions	`new ReentrantLock(true)`, multiple `Condition`s	None; one wait set
Reentrancy	`ReentrantLock` is reentrant	**Always** reentrant
Risk	Forget `unlock()` and the lock is held forever	Nothing to forget
When?	When you need advanced control	Simple cases; more readable code

Deadlock

```
static void useBoth(Object x, Object y, String name) {
    synchronized (x) {        // lock 1
        pause(100);
        synchronized (y) {    // lock 2
            System.out.println(name + " done");
        }
    }
}
public static void main(String[] args)
        throws InterruptedException {
    Thread t1 = new Thread(() -> useBoth(a, b, "T1"));
    Thread t2 = new Thread(() -> useBoth(b, a, "T2"));
    t1.setDaemon(true);
    t2.setDaemon(true);
    t1.start();
    t2.start();
    Thread.sleep(500);
    String s = t1.getState() + " " + t2.getState();
    System.out.println(s);
    long[] ids = ManagementFactory.getThreadMXBean()
            .findDeadlockedThreads();
    int n = (ids == null) ? 0 : ids.length;
    System.out.println("Deadlocked: " + n);
}
```



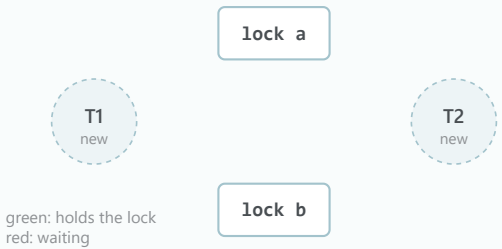
Output

```
BLOCKED BLOCKED
Deadlocked: 2
```

Two or more threads wait **forever** for locks the others hold. It happens when locks are taken **in different orders**. `pause(ms)` wraps `sleep` in try-catch.

Deadlock, Step by Step

```
▶ static final Object a = new Object(), b = new Object();
2 static void useBoth(Object x, Object y, String name) {
3     synchronized (x) {
4         pause(100);
5         synchronized (y) {
6             System.out.println(name + " done");
7         }
8     }
9 }
▶ public static void main(String[] args) {
11    new Thread(() -> useBoth(a, b, "T1")).start();
12    new Thread(() -> useBoth(b, a, "T2")).start();
13 }
```



There are two lock objects; nobody holds them yet.

Step 1 / 8

VARIABLES

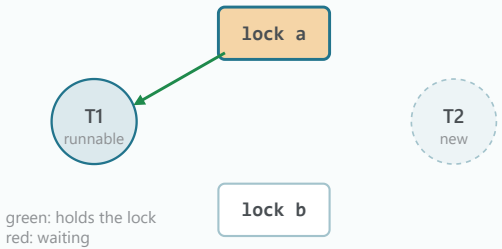
`a = Object` `b = Object`

CALL STACK

`main()`

Deadlock, Step by Step

```
1  static final Object a = new Object(), b = new Object();
2  static void useBoth(Object x, Object y, String name) {
▶      synchronized (x) {
4          pause(100);
5          synchronized (y) {
6              System.out.println(name + " done");
7          }
8      }
9  }
10 public static void main(String[] args) {
▶      new Thread(() -> useBoth(a, b, "T1")).start();
12     new Thread(() -> useBoth(b, a, "T2")).start();
13 }
```



green: holds the lock
red: waiting

T1 starts and takes **a** with `synchronized (a)`.

Step 2 / 8

VARIABLES

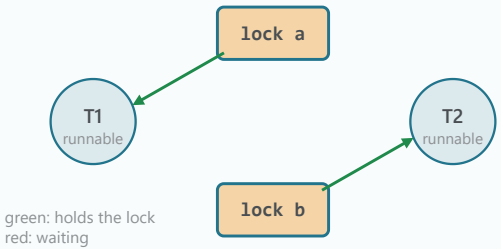
`T1: x = a, y=b`

CALL STACK

`T1: useBoth(a, b)`

Deadlock, Step by Step

```
1  static final Object a = new Object(), b = new Object();
2  static void useBoth(Object x, Object y, String name) {
▶      synchronized (x) {
4          pause(100);
5          synchronized (y) {
6              System.out.println(name + " done");
7          }
8      }
9  }
10 public static void main(String[] args) {
11     new Thread(() -> useBoth(a, b, "T1")).start();
▶      new Thread(() -> useBoth(b, a, "T2")).start();
13 }
```



T2 was called with the order reversed: it takes **b** first. No problem yet.

Step 3 / 8

VARIABLES

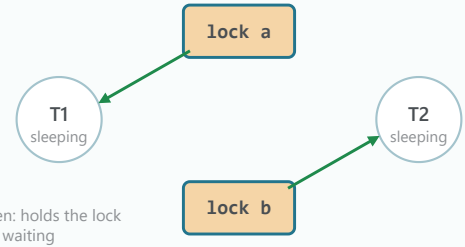
`T2: x = b, y=a`

CALL STACK

`T2: useBoth(b, a)`

Deadlock, Step by Step

```
1  static final Object a = new Object(), b = new Object();
2  static void useBoth(Object x, Object y, String name) {
3      synchronized (x) {
4          pause(100);
5          synchronized (y) {
6              System.out.println(name + " done");
7          }
8      }
9  }
10 public static void main(String[] args) {
11     new Thread(() -> useBoth(a, b, "T1")).start();
12     new Thread(() -> useBoth(b, a, "T2")).start();
13 }
```



Both sleep for 100 ms; `sleep` does **not release** the locks held.

Step 4 / 8

VARIABLES

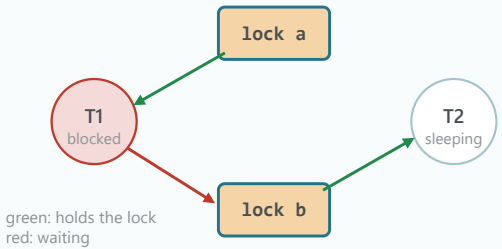
`a` = **held by T1** `b` = **held by T2**

CALL STACK

- **T2: pause(100)**
- T1: pause(100)

Deadlock, Step by Step

```
1  static final Object a = new Object(), b = new Object();
2  static void useBoth(Object x, Object y, String name) {
3      synchronized (x) {
4          pause(100);
▶          synchronized (y) {
6              System.out.println(name + " done");
7          }
8      }
9  }
10 public static void main(String[] args) {
11     new Thread(() -> useBoth(a, b, "T1")).start();
12     new Thread(() -> useBoth(b, a, "T2")).start();
13 }
```



T1 wakes up and asks for **b** for the inner block. T2 has b: T1 is `BLOCKED`.

Step 5 / 8

VARIABLES

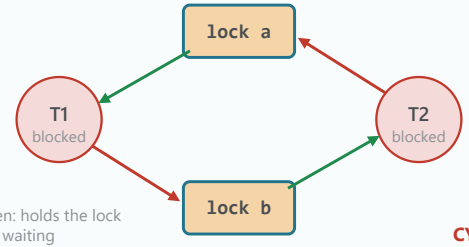
`T1 wants = b`

CALL STACK

`T1: useBoth(a, b)`

Deadlock, Step by Step

```
1  static final Object a = new Object(), b = new Object();
2  static void useBoth(Object x, Object y, String name) {
3      synchronized (x) {
4          pause(100);
▶          synchronized (y) {
6              System.out.println(name + " done");
7          }
8      }
9  }
10 public static void main(String[] args) {
11     new Thread(() -> useBoth(a, b, "T1")).start();
12     new Thread(() -> useBoth(b, a, "T2")).start();
13 }
```



T2 asks for **a**; T1 has a. Each waits for the other to let go: a **cycle** has formed.

Step 6 / 8

VARIABLES

`T2 wants = a`

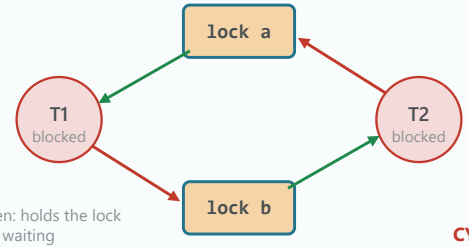
CALL STACK

`T2: useBoth(b, a)`

`T1: useBoth(a, b)`

Deadlock, Step by Step

```
1  static final Object a = new Object(), b = new Object();
2  static void useBoth(Object x, Object y, String name) {
3      synchronized (x) {
4          pause(100);
5          synchronized (y) {
▶              System.out.println(name + " done");
7          }
8      }
9  }
10 public static void main(String[] args) {
11     new Thread(() -> useBoth(a, b, "T1")).start();
12     new Thread(() -> useBoth(b, a, "T2")).start();
13 }
```



Neither reaches the "done" line; the program waits forever. `findDeadlockedThreads()` reports 2 threads.

Step 7 / 8

VARIABLES

`T1 = BLOCKED` `T2 = BLOCKED`

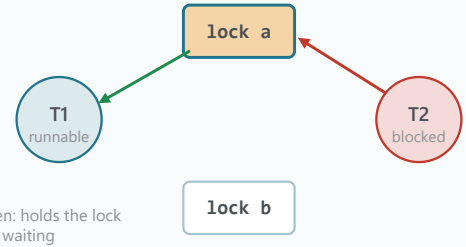
CALL STACK

`T2: useBoth(b, a)`

`T1: useBoth(a, b)`

Deadlock, Step by Step

```
1  static final Object a = new Object(), b = new Object();
2  static void useBoth(Object x, Object y, String name) {
3      synchronized (x) {
4          pause(100);
5          synchronized (y) {
6              System.out.println(name + " done");
7          }
8      }
9  }
10 public static void main(String[] args) {
11     new Thread(() -> useBoth(a, b, "T1")).start();
▶      new Thread(() -> useBoth(b, a, "T2")).start();
13 }
```



Had both threads taken the locks **in the same order** (a first), T2 would wait at a without ever holding b: no cycle could form.

Step 8 / 8

VARIABLES

`T2 call = useBoth(a, b, "T2")`

CALL STACK

- **T2: useBoth(a, b)**
- T1: useBoth(a, b)

Preventing Deadlock

ONE CHANGE: THE SAME ORDER

```
Thread t1 = new Thread(() -> useBoth(a, b, "T1"));
Thread t2 = new Thread(() -> useBoth(a, b, "T2"));
```

Output

```
T1 done
T2 done
TERMINATED TERMINATED
Deadlocked: 0
```

Everyone takes `a` first, then `b`: no cycle can form. T2 waits for `a`; it gets it once T1 is done, and finishes too.

A fixed lock order

All threads take locks **in the same order** (say, by ID).

Timed attempts

If `tryLock(time)` fails, release what you hold and retry later.

Short critical sections

Don't request another lock or do long work while holding one.

Diagnosis

`jstack` or `ThreadMXBean.findDeadlockedThreads()` reveals deadlocks.

Livelock and Starvation

Deadlock	Livelock	Starvation
everyone blocked, nothing changes	everyone busy, but no progress	one thread never gets a turn

Livelock

The threads aren't blocked; they keep changing state by "giving way" to each other, but none finishes. Example: two threads that release everything when `tryLock` fails and retry **at the same moment**, colliding every time.

Starvation

A thread gets the resource late or never: others keep grabbing the lock, or long critical sections keep it waiting. Priority values (`setPriority`) are **not reliable** for fixing this.

```
Lock fair = new ReentrantLock(true);  // fair
if (lock.tryLock(50, TimeUnit.MILLISECONDS)) {
    try { /* work */ }
    finally { lock.unlock(); }
} else {
    Thread.sleep(rnd.nextInt(20)); // random
}
```

- A **fair lock** gives the turn to the longest waiter (a bit slower).
- Waiting a **random** time before retrying breaks the symmetry and prevents livelock.
- Short critical sections give everyone a turn.

Condition Synchronization

- Gives safe access to a shared resource **only when a certain condition** holds.
- Example: a consumer should take an item only when the queue is **not empty**; a producer should add only when it is **not full**.
- Controlled updates need the object's lock, plus a way to wait until the condition holds.
- But you **can't wait while holding the lock**: the thread that would change the condition can't get the lock, and everyone gets stuck.

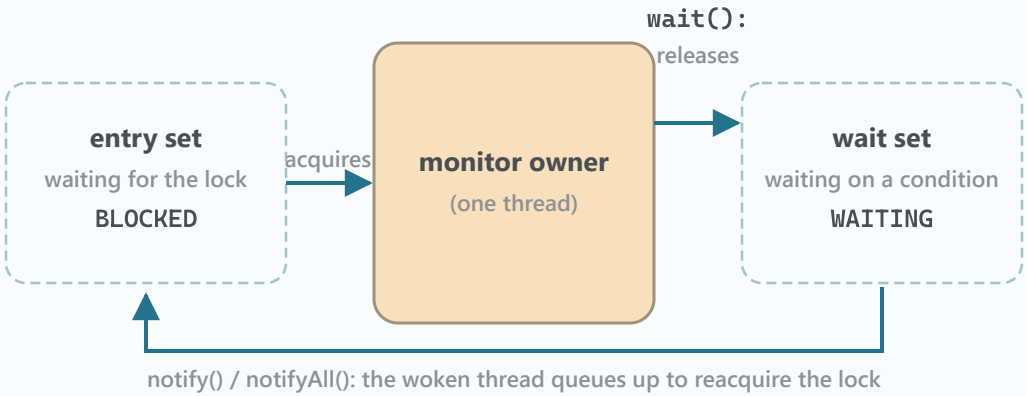
What Java provides

1) `Object.wait()` and `Object.notify()` / `notifyAll()`
2) The `Condition` interface (with `Lock`)
3) Synchronizers (`CountDownLatch`, `Semaphore`...) and the ready-made `BlockingQueue`

WRONG: BUSY WAITING

```
while (queue.isEmpty()) {
    // burns CPU; if the lock is held,
    // the producer can never add
}
```

Object.wait() and Object.notify()



- Every object has a **wait set**.
- The thread holding the lock calls `wait()`: it enters the set, **releases the lock** and blocks.
- `notify()` wakes one thread, `notifyAll()` wakes them all; `wait(ms)` also returns when time is up.
- The woken thread continues **where it left off** once it reacquires the lock.

- Rule 1: call them only inside `synchronized`; otherwise `IllegalMonitorStateException: current thread is not owner`.
- Rule 2: check the condition with `while`, not `if`: on waking it may be false again (or the wakeup may be spurious).

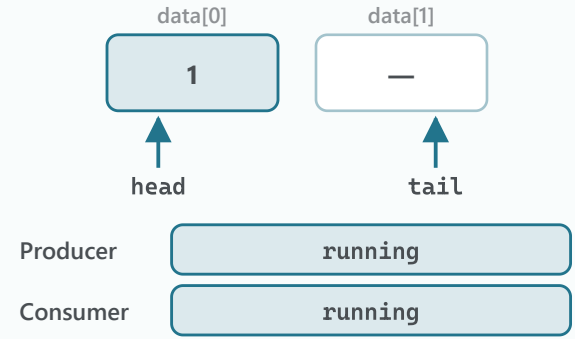
BlockingQueue with wait() and notifyAll()

```
class BlockingQueue {
    private Object[] data;
    private int head = 0;
    private int tail = 0;
    private int count = 0;
    public BlockingQueue(int capacity) {
        if (capacity <= 0)
            throw new IllegalArgumentException(
                    "capacity must be > 0");
        data = new Object[capacity];
    }
    public synchronized int getSize() {
        return count;
    }
    public synchronized void add(Object item)
            throws InterruptedException {
        while (count == data.length) {
            wait();
        }
        data[tail] = item;
        tail = (tail + 1) % data.length;
        count++;
        notifyAll();
    }
    public synchronized Object remove()
            throws InterruptedException {
        while (count == 0) {
            wait();
        }
        Object item = data[head];
        data[head] = null;
        head = (head + 1) % data.length;
        count--;
        notifyAll();
        return item;
    }
}
```

- A circular array: `head` is where to remove, `tail` where to add; `% data.length` wraps around.
- `add` waits while full, `remove` while empty; after each change `notifyAll()` wakes the waiters.

Producer–Consumer, Step by Step

```
1  public synchronized void add(Object item)
2          throws InterruptedException {
3      while (count == data.length) {
4          wait();
5      }
▶      data[tail] = item;
▶      tail = (tail + 1) % data.length;
▶      count++;
9      notifyAll();
10 }
11 public synchronized Object remove()
12         throws InterruptedException {
13     while (count == 0) {
14         wait();
15     }
16     Object item = data[head];
17     data[head] = null;
18     head = (head + 1) % data.length;
19     count--;
20     notifyAll();
21     return item;
22 }
```



The producer calls `add(1)`: there is room, so 1 goes in; `tail` moves to 1.

Step 1 / 9

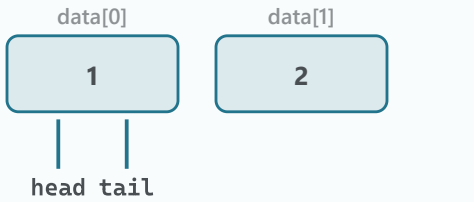
VARIABLES

`head = 0` `tail = 1` `count = 1`

Capacity 2. This is one possible order; the real one depends on the scheduler, but in no order does the queue overflow or get read while empty.

Producer–Consumer, Step by Step

```
1  public synchronized void add(Object item)
2          throws InterruptedException {
3      while (count == data.length) {
4          wait();
5      }
▶      data[tail] = item;
▶      tail = (tail + 1) % data.length;
▶      count++;
9      notifyAll();
10 }
11 public synchronized Object remove()
12         throws InterruptedException {
13     while (count == 0) {
14         wait();
15     }
16     Object item = data[head];
17     data[head] = null;
18     head = (head + 1) % data.length;
19     count--;
20     notifyAll();
21     return item;
22 }
```



`add(2)` : `tail` wraps to 2 % 2 = 0. The queue is full.

Step 2 / 9

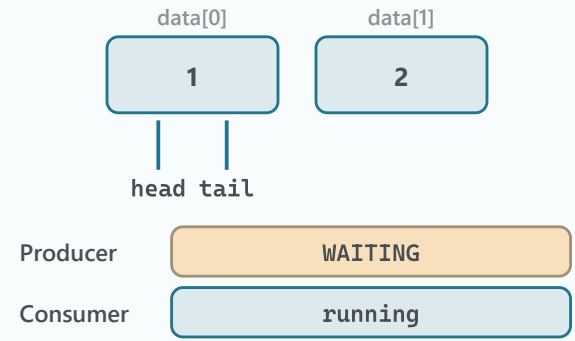
VARIABLES

`head = 0` `tail = 0` `count = 2`

Capacity 2. This is one possible order; the real one depends on the scheduler, but in no order does the queue overflow or get read while empty.

Producer–Consumer, Step by Step

```
1  public synchronized void add(Object item)
2          throws InterruptedException {
3      while (count == data.length) {
▶          wait();
5      }
6      data[tail] = item;
7      tail = (tail + 1) % data.length;
8      count++;
9      notifyAll();
10 }
11 public synchronized Object remove()
12         throws InterruptedException {
13     while (count == 0) {
14         wait();
15     }
16     Object item = data[head];
17     data[head] = null;
18     head = (head + 1) % data.length;
19     count--;
20     notifyAll();
21     return item;
22 }
```



`add(3)` : the queue is full, so `wait()` : the producer enters the wait set and **releases** the lock.

Step 3 / 9

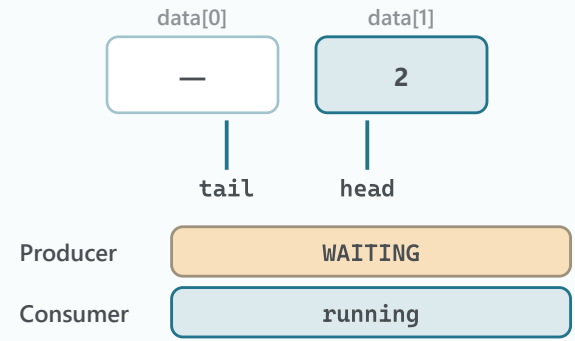
VARIABLES

`head = 0` `tail = 0` `count = 2`

Capacity 2. This is one possible order; the real one depends on the scheduler, but in no order does the queue overflow or get read while empty.

Producer–Consumer, Step by Step

```
1  public synchronized void add(Object item)
2          throws InterruptedException {
3      while (count == data.length) {
4          wait();
5      }
6      data[tail] = item;
7      tail = (tail + 1) % data.length;
8      count++;
9      notifyAll();
10 }
11 public synchronized Object remove()
12         throws InterruptedException {
13     while (count == 0) {
14         wait();
15     }
▶      Object item = data[head];
▶      data[head] = null;
▶      head = (head + 1) % data.length;
▶      count--;
20     notifyAll();
21     return item;
22 }
```



The consumer gets the lock: `remove()` takes 1, nulls the slot, and `head` becomes 1.

Step 4 / 9

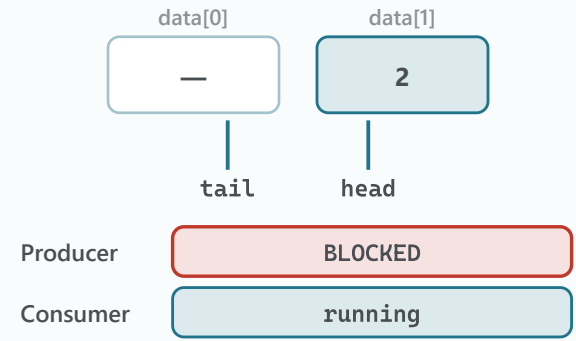
VARIABLES

`head = 1` `tail = 0` `count = 1`

Capacity 2. This is one possible order; the real one depends on the scheduler, but in no order does the queue overflow or get read while empty.

Producer–Consumer, Step by Step

```
1  public synchronized void add(Object item)
2          throws InterruptedException {
3      while (count == data.length) {
4          wait();
5      }
6      data[tail] = item;
7      tail = (tail + 1) % data.length;
8      count++;
9      notifyAll();
10 }
11 public synchronized Object remove()
12         throws InterruptedException {
13     while (count == 0) {
14         wait();
15     }
16     Object item = data[head];
17     data[head] = null;
18     head = (head + 1) % data.length;
19     count--;
▶      notifyAll();
21     return item;
22 }
```



`notifyAll()`: the producer wakes, but the consumer still holds the lock, so it waits for it. `remove` returns 1.

Step 5 / 9

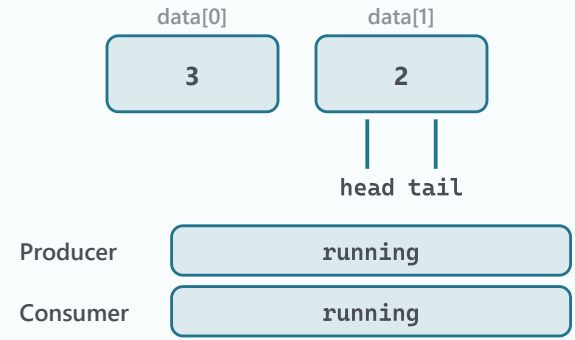
VARIABLES

`head = 1` `tail = 0` `count = 1`

Capacity 2. This is one possible order; the real one depends on the scheduler, but in no order does the queue overflow or get read while empty.

Producer–Consumer, Step by Step

```
1  public synchronized void add(Object item)
2          throws InterruptedException {
▶      while (count == data.length) {
4          wait();
5      }
▶      data[tail] = item;
▶      tail = (tail + 1) % data.length;
▶      count++;
9      notifyAll();
10 }
11 public synchronized Object remove()
12         throws InterruptedException {
13     while (count == 0) {
14         wait();
15     }
16     Object item = data[head];
17     data[head] = null;
18     head = (head + 1) % data.length;
19     count--;
20     notifyAll();
21     return item;
22 }
```



The producer gets the lock and checks the `while` condition **again**: not full. 3 is written to `data[0]`.

Step 6 / 9

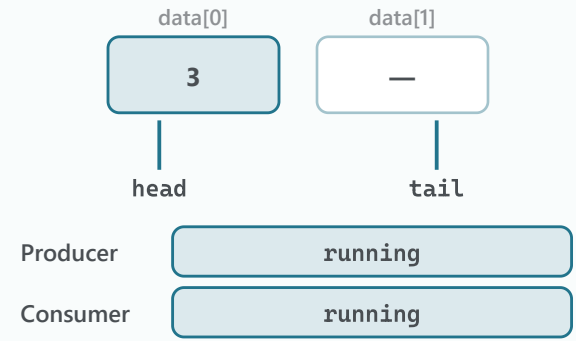
VARIABLES

`head = 1` `tail = 1` `count = 2`

Capacity 2. This is one possible order; the real one depends on the scheduler, but in no order does the queue overflow or get read while empty.

Producer–Consumer, Step by Step

```
1  public synchronized void add(Object item)
2          throws InterruptedException {
3      while (count == data.length) {
4          wait();
5      }
6      data[tail] = item;
7      tail = (tail + 1) % data.length;
8      count++;
9      notifyAll();
10 }
11 public synchronized Object remove()
12         throws InterruptedException {
13     while (count == 0) {
14         wait();
15     }
▶      Object item = data[head];
▶      data[head] = null;
▶      head = (head + 1) % data.length;
▶      count--;
20     notifyAll();
21     return item;
22 }
```



The consumer takes 2; `head` wraps around.

Step 7 / 9

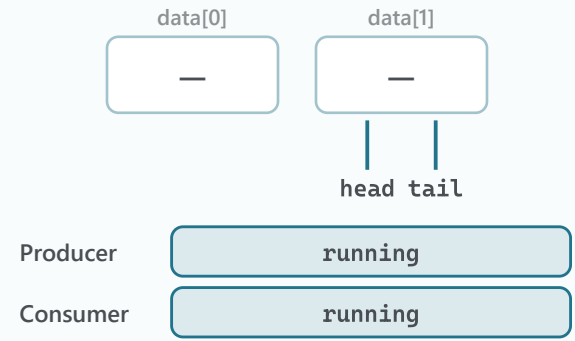
VARIABLES

`head = 0` `tail = 1` `count = 1`

Capacity 2. This is one possible order; the real one depends on the scheduler, but in no order does the queue overflow or get read while empty.

Producer–Consumer, Step by Step

```
1  public synchronized void add(Object item)
2          throws InterruptedException {
3      while (count == data.length) {
4          wait();
5      }
6      data[tail] = item;
7      tail = (tail + 1) % data.length;
8      count++;
9      notifyAll();
10 }
11 public synchronized Object remove()
12         throws InterruptedException {
13     while (count == 0) {
14         wait();
15     }
▶      Object item = data[head];
▶      data[head] = null;
▶      head = (head + 1) % data.length;
▶      count--;
20     notifyAll();
21     return item;
22 }
```



The consumer takes 3. The queue is empty.

Step 8 / 9

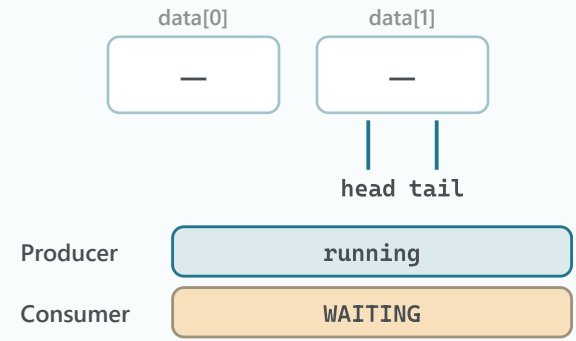
VARIABLES

`head = 1` `tail = 1` `count = 0`

Capacity 2. This is one possible order; the real one depends on the scheduler, but in no order does the queue overflow or get read while empty.

Producer–Consumer, Step by Step

```
1  public synchronized void add(Object item)
2          throws InterruptedException {
3      while (count == data.length) {
4          wait();
5      }
6      data[tail] = item;
7      tail = (tail + 1) % data.length;
8      count++;
9      notifyAll();
10 }
11 public synchronized Object remove()
12         throws InterruptedException {
13     while (count == 0) {
▶          wait();
15     }
16     Object item = data[head];
17     data[head] = null;
18     head = (head + 1) % data.length;
19     count--;
20     notifyAll();
21     return item;
22 }
```



The next `remove()`: the queue is empty, so the consumer waits; it will wake when the producer adds 4 and calls `notifyAll()`.

Step 9 / 9

VARIABLES

`head = 1` `tail = 1` `count = 0`

Capacity 2. This is one possible order; the real one depends on the scheduler, but in no order does the queue overflow or get read while empty.

Producer–Consumer: A Real Run

```
BlockingQueue queue = new BlockingQueue(2);
Thread producer = new Thread(() -> {
    try {
        for (int i = 1; i <= 4; i++) {
            queue.add(i);
            System.out.println("produced: " + i);
        }
    } catch (InterruptedException e) { }
});
Thread consumer = new Thread(() -> {
    try {
        for (int i = 1; i <= 4; i++) {
            Thread.sleep(100);
            Object x = queue.remove();
            System.out.println("consumed: " + x);
        }
    } catch (InterruptedException e) { }
});
producer.start();
consumer.start();
producer.join();
consumer.join();
System.out.println("left: " + queue.getSize());
```

Output (one run)

```
produced: 1
produced: 2
produced: 3
consumed: 1
produced: 4
consumed: 2
consumed: 3
consumed: 4
left: 0
```

The order of the lines may vary. Here “produced: 3” appeared before “consumed: 1”: the producer ran after the consumer returned from `remove` but before it printed.

The Condition Interface

- Lets a thread **wait until a state changes** and then continue where it left off.
- Brings `wait()` / `notify()` to the `Lock` world; used with a `ReentrantLock`: `lock.newCondition()`.
- `await()`: releases the lock and suspends the thread.
- `signal()` / `signalAll()`: wakes one / all waiting threads.
- A lock can have **several conditions**: producers and consumers wait in separate queues.

```
class BlockingQueue {
    private final Lock lock = new ReentrantLock();
    private final Condition notFull =
            lock.newCondition();
    private final Condition notEmpty =
            lock.newCondition();
    private Object[] data;
    private int head = 0;
    private int tail = 0;
    private int count = 0;
    public BlockingQueue(int capacity) {
        if (capacity <= 0)
            throw new IllegalArgumentException(
                    "capacity must be > 0");
        data = new Object[capacity];
    }
    // add() and remove()
}
```

BlockingQueue with Condition

```
public void add(Object item)
        throws InterruptedException {
    lock.lock();
    try {
        while (count == data.length)
            notFull.await();
        data[tail] = item;
        tail = (tail + 1) % data.length;
        count++;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

```
public Object remove()
        throws InterruptedException {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        Object item = data[head];
        data[head] = null;
        head = (head + 1) % data.length;
        count--;
        notFull.signal();
        return item;
    } finally {
        lock.unlock();
    }
}
```

Output: the producer adds 1–4, main removes 4 times

```
1 2 3 4
```

Adding signals only `notEmpty`, removing only `notFull`: no needless wakeups, so `signal()` is enough.

Ready-Made: java.util.concurrent

```
BlockingQueue<Integer> queue =
        new ArrayBlockingQueue<>(2);
Thread producer = new Thread(() -> {
    try {
        // put(): waits if the queue is full
        for (int i = 1; i <= 4; i++) queue.put(i);
    } catch (InterruptedException e) { }
});
producer.start();
for (int i = 1; i <= 4; i++) {
    // take(): waits if the queue is empty
    System.out.print(queue.take() + " ");
}
System.out.println();
producer.join();
```

Output

```
1 2 3 4
```

Approach	Waiting	Waking
`synchronized`	`wait()`	`notify()`, `notifyAll()`
`Lock` + `Condition`	`await()`	`signal()`, `signalAll()`
`ArrayBlockingQueue`	inside `put()`, `take()`	automatic

Real projects use the ready-made `java.util.concurrent.BlockingQueue` implementations (`ArrayBlockingQueue` , `LinkedBlockingQueue`); our own class is for understanding the mechanism.

Synchronizers

Ready-made tools (`java.util.concurrent`) that bring **order** among threads; each solves a different scenario.



CountDownLatch

“Let N events happen, then go on.” Waiters stop until the count reaches zero. Single-use.

CyclicBarrier

“Let's all reach the meeting point, then continue together.” Reusable.

Exchanger

Two threads meet at a point and **swap data**.

Semaphore

N permits: at most N threads can use the resource at once.

CountDownLatch

```
1  public static void main(String[] args)
2          throws Exception {
3      int count = 3;
▶      CountDownLatch latch = new CountDownLatch(count);
5      for (int i = 0; i < count; i++) {
6          int no = i + 1;   // copy for the lambda
7          new Thread(() -> {
8              System.out.println(no + ": event happened");
9              // count down when the event happens
10             latch.countDown();
11         }).start();
12     }
13     // wait for the threads to finish
14     latch.await();
15     System.out.println("All events done!");
16 }
```

Single-use: the count never goes up; `countDown()` at zero has no effect.



Synchronizes threads that wait for a number of events. **count**: the starting count, here 3.

Step 1 / 7

OUTPUT

CountDownLatch

```
1  public static void main(String[] args)
2          throws Exception {
3      int count = 3;
4      CountDownLatch latch = new CountDownLatch(count);
5      for (int i = 0; i < count; i++) {
6          int no = i + 1;   // copy for the lambda
7          new Thread(() -> {
8              System.out.println(no + ": event happened");
9              // count down when the event happens
10             latch.countDown();
▶          }).start();
12     }
13     // wait for the threads to finish
14     latch.await();
15     System.out.println("All events done!");
16 }
```

Single-use: the count never goes up; `countDown()` at zero has no effect.



Three threads are started; when they will run is unknown.

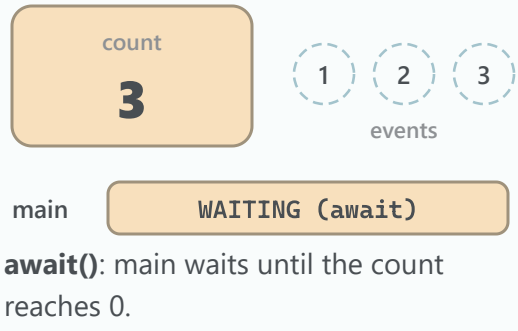
Step 2 / 7

OUTPUT

CountDownLatch

```
1  public static void main(String[] args)
2          throws Exception {
3      int count = 3;
4      CountDownLatch latch = new CountDownLatch(count);
5      for (int i = 0; i < count; i++) {
6          int no = i + 1;   // copy for the lambda
7          new Thread(() -> {
8              System.out.println(no + ": event happened");
9              // count down when the event happens
10             latch.countDown();
11         }).start();
12     }
13     // wait for the threads to finish
▶      latch.await();
15     System.out.println("All events done!");
16 }
```

Single-use: the count never goes up; `countDown()` at zero has no effect.



await(): main waits until the count reaches 0.

Step 3 / 7

OUTPUT

CountDownLatch

```
1  public static void main(String[] args)
2          throws Exception {
3      int count = 3;
4      CountDownLatch latch = new CountDownLatch(count);
5      for (int i = 0; i < count; i++) {
6          int no = i + 1;   // copy for the lambda
7          new Thread(() -> {
▶              System.out.println(no + ": event happened");
9              // count down when the event happens
▶              latch.countDown();
11         }).start();
12     }
13     // wait for the threads to finish
14     latch.await();
15     System.out.println("All events done!");
16 }
```

Single-use: the count never goes up; `countDown()` at zero has no effect.



Thread 1 prints; **countDown()** lowers the count by 1: 2.

Step 4 / 7

OUTPUT

```
1: event happened
```

CountDownLatch

```
1  public static void main(String[] args)
2          throws Exception {
3      int count = 3;
4      CountDownLatch latch = new CountDownLatch(count);
5      for (int i = 0; i < count; i++) {
6          int no = i + 1;   // copy for the lambda
7          new Thread(() -> {
               System.out.println(no + ": event happened");
9              // count down when the event happens
               latch.countDown();
11         }).start();
12     }
13     // wait for the threads to finish
14     latch.await();
15     System.out.println("All events done!");
16 }
```



In this run, thread 3 came next: the count is 1.

Step 5 / 7

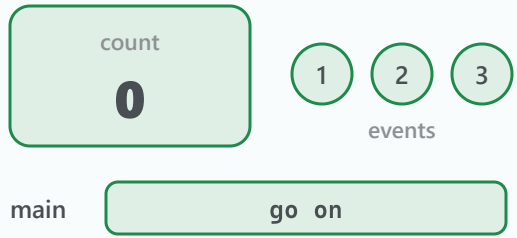
OUTPUT

```
1: event happened
3: event happened
```

Single-use: the count never goes up; `countDown()` at zero has no effect.

CountDownLatch

```
1  public static void main(String[] args)
2          throws Exception {
3      int count = 3;
4      CountDownLatch latch = new CountDownLatch(count);
5      for (int i = 0; i < count; i++) {
6          int no = i + 1;   // copy for the lambda
7          new Thread(() -> {
               System.out.println(no + ": event happened");
9              // count down when the event happens
               latch.countDown();
11         }).start();
12     }
13     // wait for the threads to finish
14     latch.await();
15     System.out.println("All events done!");
16 }
```



The last event: the count is 0. The latch opens and main wakes up.

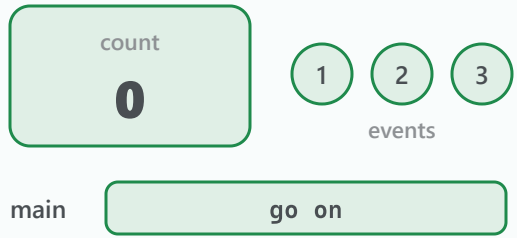
OUTPUT

```
1: event happened
3: event happened
2: event happened
```

Single-use: the count never goes up; `countDown()` at zero has no effect.

CountDownLatch

```
1  public static void main(String[] args)
2          throws Exception {
3      int count = 3;
4      CountDownLatch latch = new CountDownLatch(count);
5      for (int i = 0; i < count; i++) {
6          int no = i + 1;   // copy for the lambda
7          new Thread(() -> {
8              System.out.println(no + ": event happened");
9              // count down when the event happens
10             latch.countDown();
11         }).start();
12     }
13     // wait for the threads to finish
14     latch.await();
▶      System.out.println("All events done!");
16 }
```



This line always comes last; the order of the event lines may vary.

Step 7 / 7

OUTPUT

```
1: event happened
3: event happened
2: event happened
All events done!
```

Single-use: the count never goes up; `countDown()` at zero has no effect.

CyclicBarrier

```
public static void main(String[] args) {
    int parties = 3;
    Runnable action = () -> {
        System.out.println("Party complete!");
    };
    CyclicBarrier barrier =
            new CyclicBarrier(parties, action);
    for (int i = 0; i < parties; i++) {
        int no = i + 1;
        new Thread(() -> {
            try {
                System.out.println(no + " waiting");
                // wait for the others
                barrier.await();
                System.out.println(no + " continuing");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

- Makes a group of threads wait at a **meeting point** (the barrier); **parties**: how many to wait for.
- `await()` : wait until the party is complete. Then the **barrier action** runs once and everyone continues.
- An exception, a timeout or an interrupt **breaks** the barrier: the waiters get `BrokenBarrierException` .

Output (one run)

```
1 waiting
2 waiting
3 waiting
Party complete!
3 continuing
1 continuing
2 continuing
```

Exchanger

```
static void swap(Exchanger<String> ex, String data, int ms) {
    try {
        System.out.println("sending: " + data);
        Thread.sleep(ms);
        String got = ex.exchange(data);   // wait for the other
        System.out.println(data + " owner got: " + got);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
public static void main(String[] args) {
    Exchanger<String> exchanger = new Exchanger<>();
    // The first thread sends "Data 1" after 2 s
    new Thread(() -> swap(exchanger, "Data 1", 2000)).start();
    // The second thread sends "Data 2" after 1 s
    new Thread(() -> swap(exchanger, "Data 2", 1000)).start();
```

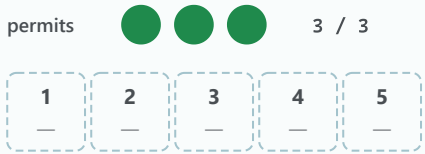
Output

```
sending: Data 2
sending: Data 1
Data 1 owner got: Data 2
Data 2 owner got: Data 1
```

- Used by two threads to **exchange data**; more a communication object than a lock.
- `exchange(data)` **waits** until the other side also calls `exchange` : the “Data 2” owner waited 1 s.
- When they meet, each gets the other's data.

Semaphore

```
1  public static void main(String[] args) {
2      int permits = 3;
▶      Semaphore semaphore = new Semaphore(permits);
4      for (int i = 0; i < 5; i++) {
5          int no = i + 1;
6          new Thread(() -> {
7              try {
8                  semaphore.acquire();   // take a
   permit
9                  System.out.println(no + " in");
10                 Thread.sleep(2000);
11                 System.out.println(no + " out");
12                 semaphore.release();   // give it
   back
13             } catch (InterruptedException e) {
14                 e.printStackTrace();
15             }
16         }).start();
17     }
18 }
```



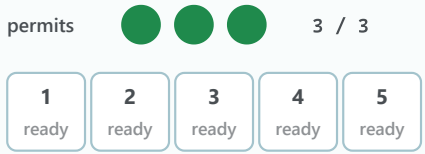
A semaphore limits concurrent access: here at most 3 threads at a time. `new Semaphore(n, true)` : fair distribution.

Step 1 / 10

OUTPUT

Semaphore

```
1  public static void main(String[] args) {
2      int permits = 3;
3      Semaphore semaphore = new Semaphore(permits);
4      for (int i = 0; i < 5; i++) {
5          int no = i + 1;
6          new Thread(() -> {
7              try {
8                  semaphore.acquire();   // take a
   permit
9                  System.out.println(no + " in");
10                 Thread.sleep(2000);
11                 System.out.println(no + " out");
12                 semaphore.release();   // give it
   back
13             } catch (InterruptedException e) {
14                 e.printStackTrace();
15             }
▶          }).start();
17     }
18 }
```



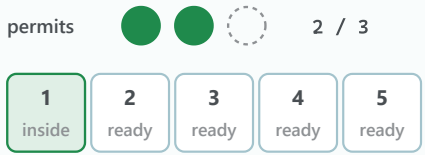
Five threads start. `acquire()` waits if no permit is left; `release()` gives the permit back.

Step 2 / 10

OUTPUT

Semaphore

```
1  public static void main(String[] args) {
2      int permits = 3;
3      Semaphore semaphore = new Semaphore(permits);
4      for (int i = 0; i < 5; i++) {
5          int no = i + 1;
6          new Thread(() -> {
7              try {
               semaphore.acquire();   // take a permit
               System.out.println(no + " in");
10             Thread.sleep(2000);
11             System.out.println(no + " out");
12             semaphore.release();   // give it back
13         } catch (InterruptedException e) {
14             e.printStackTrace();
15         }
16     }).start();
17 }
18 }
```



1 takes a permit and enters.

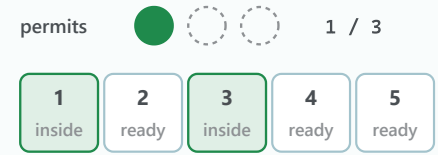
Step 3 / 10

OUTPUT

```
1 in
```

Semaphore

```
1  public static void main(String[] args) {
2      int permits = 3;
3      Semaphore semaphore = new Semaphore(permits);
4      for (int i = 0; i < 5; i++) {
5          int no = i + 1;
6          new Thread(() -> {
7              try {
▶                  semaphore.acquire();   // take a permit
▶                  System.out.println(no + " in");
10                 Thread.sleep(2000);
11                 System.out.println(no + " out");
12                 semaphore.release();   // give it back
13             } catch (InterruptedException e) {
14                 e.printStackTrace();
15             }
16         }).start();
17     }
18 }
```



3 enters.

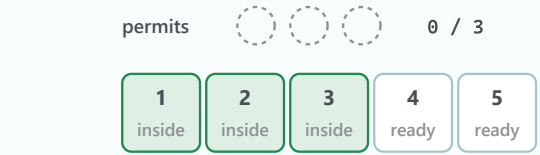
Step 4 / 10

OUTPUT

```
1 in
3 in
```

Semaphore

```
1  public static void main(String[] args) {
2      int permits = 3;
3      Semaphore semaphore = new Semaphore(permits);
4      for (int i = 0; i < 5; i++) {
5          int no = i + 1;
6          new Thread(() -> {
7              try {
               semaphore.acquire();   // take a permit
               System.out.println(no + " in");
10             Thread.sleep(2000);
11             System.out.println(no + " out");
12             semaphore.release();   // give it back
13         } catch (InterruptedException e) {
14             e.printStackTrace();
15         }
16     }).start();
17 }
18 }
```



2 enters. No permits left.

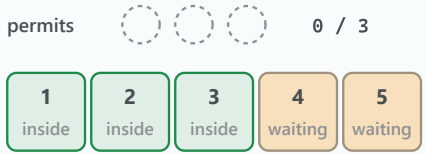
Step 5 / 10

OUTPUT

```
1 in
3 in
2 in
```

Semaphore

```
1  public static void main(String[] args) {
2      int permits = 3;
3      Semaphore semaphore = new Semaphore(permits);
4      for (int i = 0; i < 5; i++) {
5          int no = i + 1;
6          new Thread(() -> {
7              try {
               semaphore.acquire();   // take a permit
9                  System.out.println(no + " in");
10                 Thread.sleep(2000);
11                 System.out.println(no + " out");
12                 semaphore.release();   // give it back
13             } catch (InterruptedException e) {
14                 e.printStackTrace();
15             }
16         }).start();
17     }
18 }
```



4 and 5 wait in `acquire()`.

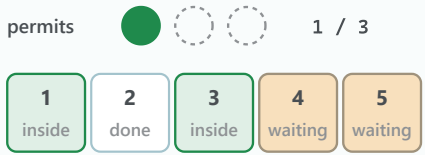
Step 6 / 10

OUTPUT

```
1 in
3 in
2 in
```

Semaphore

```
1  public static void main(String[] args) {
2      int permits = 3;
3      Semaphore semaphore = new Semaphore(permits);
4      for (int i = 0; i < 5; i++) {
5          int no = i + 1;
6          new Thread(() -> {
7              try {
8                  semaphore.acquire();   // take a permit
9                  System.out.println(no + " in");
10                 Thread.sleep(2000);
▶                  System.out.println(no + " out");
▶                  semaphore.release();   // give it back
13             } catch (InterruptedException e) {
14                 e.printStackTrace();
15             }
16         }).start();
17     }
18 }
```



After 2 s, 2 leaves and releases its permit.

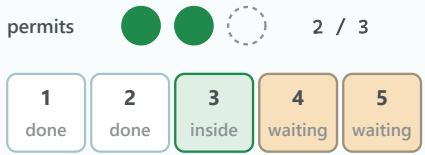
Step 7 / 10

OUTPUT

```
1 in
3 in
2 in
2 out
```

Semaphore

```
1  public static void main(String[] args) {
2      int permits = 3;
3      Semaphore semaphore = new Semaphore(permits);
4      for (int i = 0; i < 5; i++) {
5          int no = i + 1;
6          new Thread(() -> {
7              try {
8                  semaphore.acquire();   // take a
   permit
9                  System.out.println(no + " in");
10                 Thread.sleep(2000);
▶                  System.out.println(no + " out");
▶                  semaphore.release();   // give it
   back
13             } catch (InterruptedException e) {
14                 e.printStackTrace();
15             }
16         }).start();
17     }
18 }
```



1 leaves too.

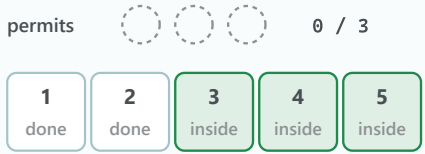
Step 8 / 10

OUTPUT

```
1 in
3 in
2 in
2 out
1 out
```

Semaphore

```
1  public static void main(String[] args) {
2      int permits = 3;
3      Semaphore semaphore = new Semaphore(permits);
4      for (int i = 0; i < 5; i++) {
5          int no = i + 1;
6          new Thread(() -> {
7              try {
               semaphore.acquire();   // take a permit
               System.out.println(no + " in");
10             Thread.sleep(2000);
11             System.out.println(no + " out");
12             semaphore.release();   // give it back
13         } catch (InterruptedException e) {
14             e.printStackTrace();
15         }
16     }).start();
17 }
18 }
```



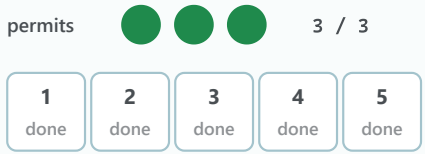
The waiting 4 and 5 take the permits and enter.

OUTPUT

```
1 in
3 in
2 in
2 out
1 out
4 in
5 in
```

Semaphore

```
1  public static void main(String[] args) {
2      int permits = 3;
3      Semaphore semaphore = new Semaphore(permits);
4      for (int i = 0; i < 5; i++) {
5          int no = i + 1;
6          new Thread(() -> {
7              try {
8                  semaphore.acquire();   // take a
   permit
9                  System.out.println(no + " in");
10                 Thread.sleep(2000);
▶                  System.out.println(no + " out");
▶                  semaphore.release();   // give it
   back
13             } catch (InterruptedException e) {
14                 e.printStackTrace();
15             }
16         }).start();
17     }
18 }
```



3, 4 and 5 leave too; all 3 permits are back.

Step 10 / 10

OUTPUT

```
1 in
3 in
2 in
2 out
1 out
4 in
5 in
3 out
4 out
5 out
```

Atomic Classes

```java
public class AtomicCounter {
    public static void main(String[] args)
            throws InterruptedException {
        AtomicInteger counter = new AtomicInteger(0);
        Runnable r = () -> {
            for (int i = 0; i < 100_000; i++) {
                counter.incrementAndGet();   // atomic ++counter
            }
        };
        Thread a = new Thread(r), b = new Thread(r);
        a.start(); b.start();
        a.join(); b.join();
        System.out.println("Result: " + counter.get());
        boolean b1 = counter.compareAndSet(200000, 0);
        System.out.println(b1 + " " + counter.get());
        boolean b2 = counter.compareAndSet(5, 10);
        System.out.println(b2 + " " + counter.get());
    }
}
```

Output

```
Result: 200000
true 0
false 0
```

- `AtomicInteger`, `AtomicLong`, `AtomicBoolean`...: safe access without locks.
- `incrementAndGet()` = an atomic `++x`; it can't be interrupted halfway.
- An atomic object ≈ a `volatile` field + an atomic `compareAndSet`.
- `compareAndSet(expected, update)`: if the value is still *expected*, writes *update* and returns `true`; otherwise leaves it alone.

Lock-Free Increment with `compareAndSet`

```
▶ AtomicInteger counter = new AtomicInteger(5);
2
3 int inc() {                    // how incrementAndGet
  works
4     while (true) {
5         int old = counter.get();
6         int next = old + 1;
7         if (counter.compareAndSet(old, next))
8             return next;
9         // failed: someone got in first; retry
10    }
11 }
```



Two threads will increment the same `counter` object. It starts at 5.

Step 1 / 9

VARIABLES

`counter = 5`

CALL STACK

empty

Lock-Free Increment with `compareAndSet`

```
1  AtomicInteger counter = new AtomicInteger(5);
2
3  int inc() {                    // how incrementAndGet
   works
4      while (true) {
▶          int old = counter.get();
6          int next = old + 1;
7          if (counter.compareAndSet(old, next))
8              return next;
9          // failed: someone got in first; retry
10     }
11 }
```



T1 reads the current value: `old` = 5.

Step 2 / 9

VARIABLES

`T1.old = 5`

CALL STACK

`T1: inc()`

Lock-Free Increment with `compareAndSet`

```
1  AtomicInteger counter = new AtomicInteger(5);
2
3  int inc() {                    // how incrementAndGet
   works
4      while (true) {
▶          int old = counter.get();
6          int next = old + 1;
7          if (counter.compareAndSet(old, next))
8              return next;
9          // failed: someone got in first; retry
10     }
11 }
```



Before T1 goes on, T2 reads too and also sees 5. There's no lock, so both keep running.

Step 3 / 9

VARIABLES

`T1.old = 5` `T2.old = 5`

CALL STACK

`T2: inc()`

`T1: inc()`

Lock-Free Increment with `compareAndSet`

```
1  AtomicInteger counter = new AtomicInteger(5);
2
3  int inc() {                    // how incrementAndGet
   works
4      while (true) {
5          int old = counter.get();
▶          int next = old + 1;
7          if (counter.compareAndSet(old, next))
8              return next;
9          // failed: someone got in first; retry
10     }
11 }
```



Both compute `next` = 6 locally. With a plain `counter++`, both would write 6 and one increment would be lost.

Step 4 / 9

VARIABLES

`T1.next = 6` `T2.next = 6`

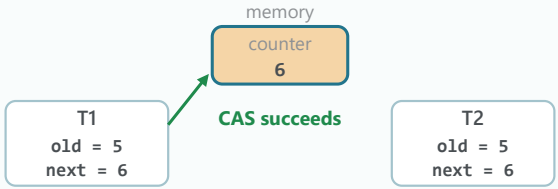
CALL STACK

`T2: inc()`

`T1: inc()`

Lock-Free Increment with `compareAndSet`

```
1  AtomicInteger counter = new AtomicInteger(5);
2
3  int inc() {                    // how incrementAndGet
   works
4      while (true) {
5          int old = counter.get();
6          int next = old + 1;
▶          if (counter.compareAndSet(old, next))
▶              return next;
9          // failed: someone got in first; retry
10     }
11 }
```



T1: `compareAndSet(5, 6)`. The value is still 5 → 6 is written and `true` returned. Check and write are **one atomic** operation.

Step 5 / 9

VARIABLES

`counter = 6`

CALL STACK

`T1: inc()`

Lock-Free Increment with `compareAndSet`

```
1  AtomicInteger counter = new AtomicInteger(5);
2
3  int inc() {                    // how incrementAndGet
   works
4      while (true) {
5          int old = counter.get();
6          int next = old + 1;
▶          if (counter.compareAndSet(old, next))
8              return next;
9          // failed: someone got in first; retry
10     }
11 }
```



T2: `compareAndSet(5, 6)`. The value is now 6, not the expected 5 → nothing is written; `false`.

Step 6 / 9

VARIABLES

`counter = 6`

`T2 CAS(5, 6) = false`

CALL STACK

`T2: inc()`

Lock-Free Increment with `compareAndSet`

```
1  AtomicInteger counter = new AtomicInteger(5);
2
3  int inc() {                    // how incrementAndGet
   works
4      while (true) {
▶          int old = counter.get();
6          int next = old + 1;
7          if (counter.compareAndSet(old, next))
8              return next;
9          // failed: someone got in first; retry
10     }
11 }
```



The loop repeats: T2 rereads the current value.

Step 7 / 9

VARIABLES

`T2.old = 6`

CALL STACK

`T2: inc()`

Lock-Free Increment with `compareAndSet`

```
1  AtomicInteger counter = new AtomicInteger(5);
2
3  int inc() {                    // how incrementAndGet
   works
4      while (true) {
5          int old = counter.get();
▶          int next = old + 1;
7          if (counter.compareAndSet(old, next))
8              return next;
9          // failed: someone got in first; retry
10     }
11 }
```



`next` = 7.

Step 8 / 9

VARIABLES

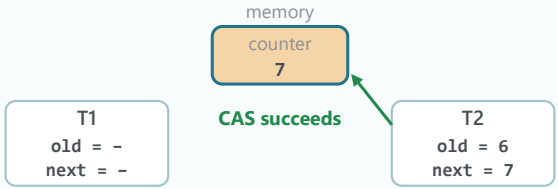
`T2.next = 7`

CALL STACK

`T2: inc()`

Lock-Free Increment with `compareAndSet`

```
1  AtomicInteger counter = new AtomicInteger(5);
2
3  int inc() {                    // how incrementAndGet
   works
4      while (true) {
5          int old = counter.get();
6          int next = old + 1;
▶          if (counter.compareAndSet(old, next))
▶              return next;
9          // failed: someone got in first; retry
10     }
11 }
```



This time the CAS succeeds: `counter` = 7. Both increments counted; nobody waited for a lock, T2 just went around once more.

Step 9 / 9

VARIABLES

`counter = 7`

CALL STACK

`T2: inc()`

Interrupting

```
public class InterruptDemo {
    public static void main(String[] args)
            throws InterruptedException {
        Thread worker = new Thread(() -> {
            try {
                while (!Thread.interrupted()) {
                    // the thread's work
                    System.out.println("working...");
                    Thread.sleep(400);
                }
            } catch (InterruptedException e) {
                System.out.println("Thread cancelled!");
            }
        });
        worker.start();
        Thread.sleep(1000);
        worker.interrupt();   // interrupt it in sleep()
        worker.join();
        System.out.println("main: done");
    }
}
```

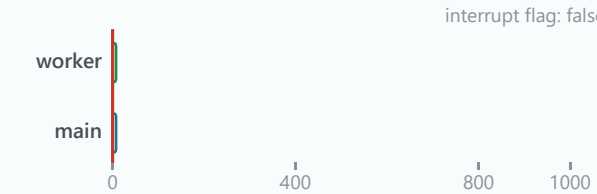
Output

```
working...
working...
working...
Thread cancelled!
main: done
```

- `interrupt()` : an **interrupt request**; if the thread is in `sleep` / `wait`, it gets an `InterruptedException`.
- `Thread.interrupted()` reads the flag and **clears** it; `isInterrupted()` only reads it.

Interrupting, Step by Step

```
▶ Thread worker = new Thread(() -> {
2     try {
3         while (!Thread.interrupted()) {
4             System.out.println("working...");
5             Thread.sleep(400);
6         }
7     } catch (InterruptedException e) {
8         System.out.println("cancelled!");
9     }
10 });
▶ worker.start();
12 Thread.sleep(1000);
13 worker.interrupt();   // worker is in sleep()
14 worker.join();
15 System.out.println("main: done");
```



`start()` sets `worker` running. The interrupt flag is `false`.

Step 1 / 9

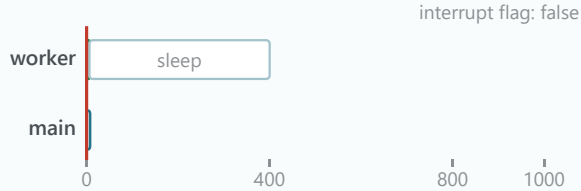
VARIABLES

`worker` = **RUNNABLE**

OUTPUT

Interrupting, Step by Step

```
 1 Thread worker = new Thread(() -> {
 2     try {
 3         while (!Thread.interrupted()) {
 4             System.out.println("working...");
 5             Thread.sleep(400);
 6         }
 7     } catch (InterruptedException e) {
 8         System.out.println("cancelled!");
 9     }
10 });
11 worker.start();
12 Thread.sleep(1000);
13 worker.interrupt();   // worker is in sleep()
14 worker.join();
15 System.out.println("main: done");
```



The flag is clear: it enters the loop, prints and sleeps 400 ms.

Step 2 / 9

VARIABLES

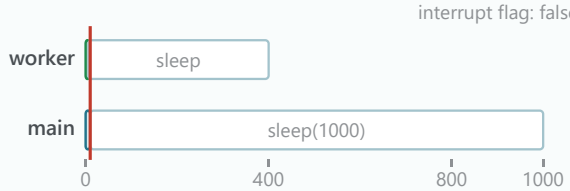
`worker = TIMED_WAITING`

OUTPUT

```
working...
```

Interrupting, Step by Step

```
1  Thread worker = new Thread(() -> {
2      try {
3          while (!Thread.interrupted()) {
4              System.out.println("working...");
5              Thread.sleep(400);
6          }
7      } catch (InterruptedException e) {
8          System.out.println("cancelled!");
9      }
10 });
11 worker.start();
▶ Thread.sleep(1000);
13 worker.interrupt();   // worker is in sleep()
14 worker.join();
15 System.out.println("main: done");
```



`main` sleeps 1000 ms too; the two threads proceed independently.

Step 3 / 9

VARIABLES

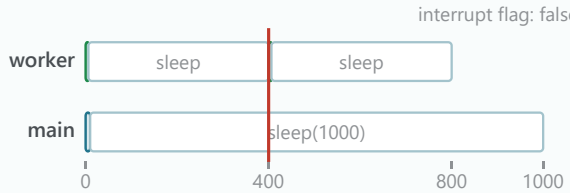
`main = TIMED_WAITING`

OUTPUT

```
working...
```

Interrupting, Step by Step

```
1  Thread worker = new Thread(() -> {
2      try {
▶          while (!Thread.interrupted()) {
▶              System.out.println("working...");
▶              Thread.sleep(400);
6          }
7      } catch (InterruptedException e) {
8          System.out.println("cancelled!");
9      }
10 });
11 worker.start();
12 Thread.sleep(1000);
13 worker.interrupt();   // worker is in sleep()
14 worker.join();
15 System.out.println("main: done");
```



worker wakes up; `Thread.interrupted()` is still `false`: one more lap.

Step 4 / 9

VARIABLES

`time = ≈400 ms`

OUTPUT

```
working...
working...
```

Interrupting, Step by Step

```
1  Thread worker = new Thread(() -> {
2      try {
▶          while (!Thread.interrupted()) {
▶              System.out.println("working...");
▶              Thread.sleep(400);
6          }
7      } catch (InterruptedException e) {
8          System.out.println("cancelled!");
9      }
10 });
11 worker.start();
12 Thread.sleep(1000);
13 worker.interrupt();   // worker is in sleep()
14 worker.join();
15 System.out.println("main: done");
```



Third lap: it prints and sleeps again (it would have slept until 1200 ms).

Step 5 / 9

VARIABLES

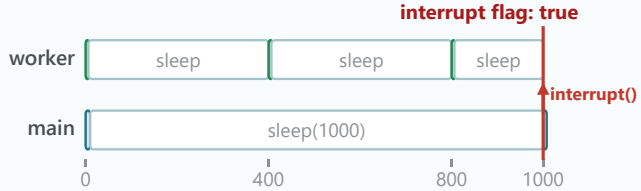
`time = ≈800 ms`

OUTPUT

```
working...
working...
working...
```

Interrupting, Step by Step

```
1  Thread worker = new Thread(() -> {
2      try {
3          while (!Thread.interrupted()) {
4              System.out.println("working...");
5              Thread.sleep(400);
6          }
7      } catch (InterruptedException e) {
8          System.out.println("cancelled!");
9      }
10 });
11 worker.start();
12 Thread.sleep(1000);
▶  worker.interrupt();   // worker is in sleep()
14 worker.join();
15 System.out.println("main: done");
```



`main` wakes up and calls `interrupt()`: worker's interrupt flag becomes `true`.

Step 6 / 9

VARIABLES

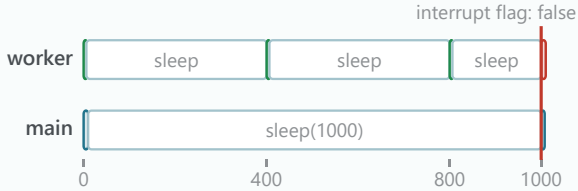
`time = ≈1000 ms`

OUTPUT

```
working...
working...
working...
```

Interrupting, Step by Step

```
1  Thread worker = new Thread(() -> {
2      try {
3          while (!Thread.interrupted()) {
4              System.out.println("working...");
5              Thread.sleep(400);
6          }
▶      } catch (InterruptedException e) {
▶          System.out.println("cancelled!");
9      }
10 });
11 worker.start();
12 Thread.sleep(1000);
13 worker.interrupt();   // worker is in sleep()
14 worker.join();
15 System.out.println("main: done");
```



worker is in `sleep`, so the sleep is cut short **at once**: an `InterruptedException` is thrown and the flag is cleared again.

Step 7 / 9

VARIABLES

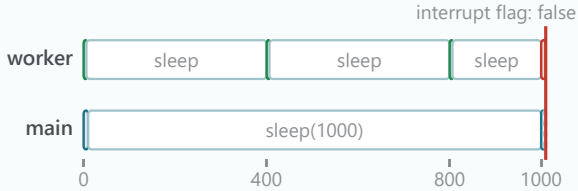
`e = InterruptedException`

OUTPUT

```
working...
working...
working...
cancelled!
```

Interrupting, Step by Step

```
1  Thread worker = new Thread(() -> {
2      try {
3          while (!Thread.interrupted()) {
4              System.out.println("working...");
5              Thread.sleep(400);
6          }
7      } catch (InterruptedException e) {
8          System.out.println("cancelled!");
9      }
10 });
11 worker.start();
12 Thread.sleep(1000);
13 worker.interrupt();   // worker is in sleep()
▶  worker.join();
15 System.out.println("main: done");
```



`main` waits for worker with `join()`; worker ends after the `catch`.

Step 8 / 9

VARIABLES

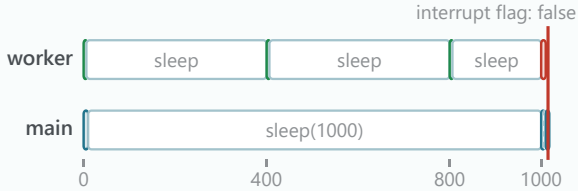
`worker = TERMINATED`

OUTPUT

```
working...
working...
working...
cancelled!
```

Interrupting, Step by Step

```
1  Thread worker = new Thread(() -> {
2      try {
3          while (!Thread.interrupted()) {
4              System.out.println("working...");
5              Thread.sleep(400);
6          }
7      } catch (InterruptedException e) {
8          System.out.println("cancelled!");
9      }
10 });
11 worker.start();
12 Thread.sleep(1000);
13 worker.interrupt();   // worker is in sleep()
14 worker.join();
▶  System.out.println("main: done");
```



The program ends cleanly. An interrupt is a **request**: the thread itself handles the flag or the exception and exits.

Step 9 / 9

VARIABLES

—

OUTPUT

```
working...
working...
working...
cancelled!
main: done
```

Suspension

```java
class Task implements Runnable {
    private boolean paused = false;
    public synchronized void pause() {
        paused = true;
    }
    public synchronized void resume() {
        paused = false;
        notifyAll();   // wake the waiter
    }
    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                synchronized (this) {
                    while (paused) wait();
                }
                System.out.println("step " + i);
                Thread.sleep(200);
            }
        } catch (InterruptedException e) { }
    }
}
```

```java
Task g = new Task();
Thread t = new Thread(g);
t.start();
Thread.sleep(500);
g.pause();
System.out.println("main: paused");
Thread.sleep(1000);
System.out.println("main: resume");
g.resume();
t.join();
```

Output

```
step 1
step 2
step 3
main: paused
main: resume
step 4
step 5
```

Use `wait()` / `notifyAll()` instead of `suspend()` / `resume()`.

The ThreadLocal Class

```
static final ThreadLocal<String> value =
        new ThreadLocal<>();
public static void main(String[] args)
        throws InterruptedException {
    Thread thread1 = new Thread(() -> {
        // first thread
        value.set("Value 1");
        System.out.println("thread1: " + value.get());
    });
    Thread thread2 = new Thread(() -> {
        // second thread
        value.set("Value 2");
        System.out.println("thread2: " + value.get());
    });
    thread1.start();
    thread2.start();
    thread1.join();
    thread2.join();
    System.out.println("main: " + value.get());
```

Run 1

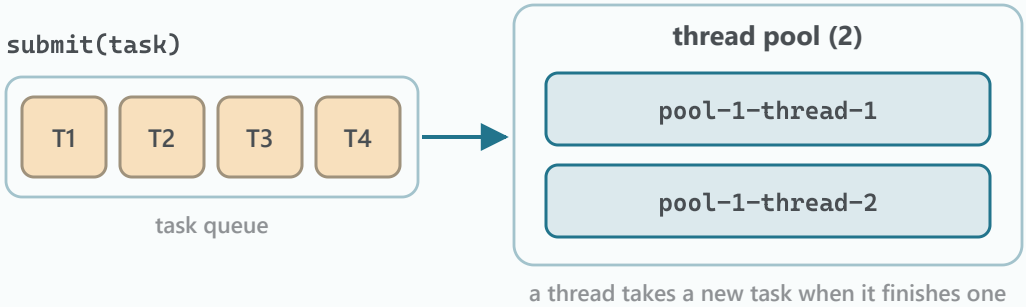
```
thread1: Value 1
thread2: Value 2
main: null
```

Run 2

```
thread2: Value 2
thread1: Value 1
main: null
```

- A variable of which each thread has **its own copy**; the copies are independent.
- main never called `set`, so it sees `null`.
- `ThreadLocal.withInitial(() -> 0)` (Java 8+), or overriding `initialValue()`, gives each thread its own initial value.

Thread Pools



- A structure that **creates threads up front** and manages them: tasks are submitted to the pool.
- Creating threads is expensive; a pool **reuses** them and caps their number.
- The `ExecutorService` interface controls the pool's work; `Executors.newFixedThreadPool(n)` gives a ready-made pool.
- `ThreadPoolExecutor`: the class that creates and manages the pool, with fine-grained settings.

ExecutorService

```java
public class PoolDemo {
    static void task(int no) {
        String name = Thread.currentThread().getName();
        System.out.println("Task " + no + " -> " + name);
    }
    public static void main(String[] args) throws Exception {
        ExecutorService pool = Executors.newFixedThreadPool(2);
        for (int i = 1; i <= 4; i++) {
            int no = i;
            pool.submit(() -> task(no));
        }
        // Callable: a task that returns a value
        Future<Integer> f = pool.submit(() -> 6 * 7);
        System.out.println("Callable result: " + f.get());
        pool.shutdown();   // accept no new tasks
        pool.awaitTermination(5, TimeUnit.SECONDS);
        System.out.println("Pool shut down");
    }
}
```

Java 21+: `Executors.newVirtualThreadPerTaskExecutor()` gives each task a lightweight **virtual thread**.

Run 1

```
Task 1 -> pool-1-thread-1
Task 2 -> pool-1-thread-2
Task 3 -> pool-1-thread-2
Task 4 -> pool-1-thread-2
Callable result: 42
Pool shut down
```

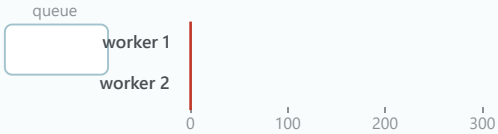
What differed in run 2

```
Task 3 -> pool-1-thread-1
Task 4 -> pool-1-thread-2
```

A `Callable` returns a value; `Future.get()` waits for the result. Forget `shutdown()` and the pool's threads keep the program alive.

The Pool and `Future`, Step by Step

```
1  static int job(int ms, int d) throws Exception {
2      Thread.sleep(ms);   // stands in for slow work
3      return d;
4  }
5  // inside main:
▶ ExecutorService pool = Executors.newFixedThreadPool(2);
7  Future<Integer> f1 = pool.submit(() -> job(300, 10));
8  Future<Integer> f2 = pool.submit(() -> job(100, 20));
9  Future<Integer> f3 = pool.submit(() -> job(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int sum = f1.get() + f2.get() + f3.get();
12 System.out.println("sum = " + sum);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 pool.shutdown();
```



main: running

done: –

A fixed pool of 2 threads. Workers run tasks as they come and take a new one when done.

Step 1 / 10

VARIABLES

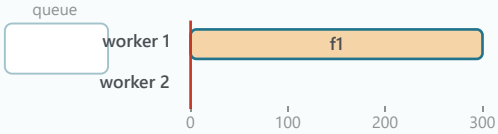
`pool = 2 workers`

OUTPUT

Times are approximate (ignoring thread start-up delay); which worker gets which task may vary from run to run.

The Pool and `Future`, Step by Step

```
1  static int job(int ms, int d) throws Exception {
2      Thread.sleep(ms);   // stands in for slow work
3      return d;
4  }
5  // inside main:
6  ExecutorService pool = Executors.newFixedThreadPool(2);
▶  Future<Integer> f1 = pool.submit(() -> job(300, 10));
8  Future<Integer> f2 = pool.submit(() -> job(100, 20));
9  Future<Integer> f3 = pool.submit(() -> job(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int sum = f1.get() + f2.get() + f3.get();
12 System.out.println("sum = " + sum);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 pool.shutdown();
```



main: running

done: –

`submit` hands the task to the pool and returns a `Future` at once. f1 starts on worker 1.

Step 2 / 10

VARIABLES

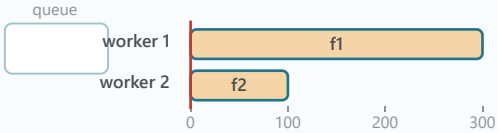
`f1 = Future (pending)`

OUTPUT

Times are approximate (ignoring thread start-up delay); which worker gets which task may vary from run to run.

The Pool and `Future`, Step by Step

```
1  static int job(int ms, int d) throws Exception {
2      Thread.sleep(ms);   // stands in for slow work
3      return d;
4  }
5  // inside main:
6  ExecutorService pool = Executors.newFixedThreadPool(2);
7  Future<Integer> f1 = pool.submit(() -> job(300, 10));
▶ Future<Integer> f2 = pool.submit(() -> job(100, 20));
9  Future<Integer> f3 = pool.submit(() -> job(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int sum = f1.get() + f2.get() + f3.get();
12 System.out.println("sum = " + sum);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 pool.shutdown();
```



main: running

done: –

f2 goes to the idle worker 2.

Step 3 / 10

VARIABLES

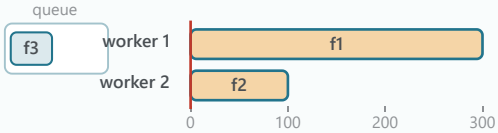
`f2 = Future`

OUTPUT

Times are approximate (ignoring thread start-up delay); which worker gets which task may vary from run to run.

The Pool and `Future`, Step by Step

```
1  static int job(int ms, int d) throws Exception {
2      Thread.sleep(ms);   // stands in for slow work
3      return d;
4  }
5  // inside main:
6  ExecutorService pool = Executors.newFixedThreadPool(2);
7  Future<Integer> f1 = pool.submit(() -> job(300, 10));
8  Future<Integer> f2 = pool.submit(() -> job(100, 20));
▶  Future<Integer> f3 = pool.submit(() -> job(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int sum = f1.get() + f2.get() + f3.get();
12 System.out.println("sum = " + sum);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 pool.shutdown();
```



main: running

done: –

Both workers are busy: f3 waits **in the queue**. No new thread is created.

Step 4 / 10

VARIABLES

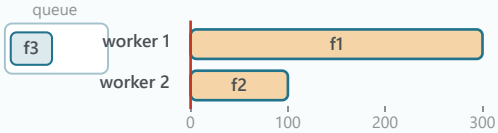
`f3 = Future`

OUTPUT

Times are approximate (ignoring thread start-up delay); which worker gets which task may vary from run to run.

The Pool and `Future`, Step by Step

```
1  static int job(int ms, int d) throws Exception {
2      Thread.sleep(ms);   // stands in for slow work
3      return d;
4  }
5  // inside main:
6  ExecutorService pool = Executors.newFixedThreadPool(2);
7  Future<Integer> f1 = pool.submit(() -> job(300, 10));
8  Future<Integer> f2 = pool.submit(() -> job(100, 20));
9  Future<Integer> f3 = pool.submit(() -> job(100, 30));
▶ System.out.println(f1.isDone() + " " + f3.isDone());
11 int sum = f1.get() + f2.get() + f3.get();
12 System.out.println("sum = " + sum);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 pool.shutdown();
```



main: running

done: –

No task is done yet; `isDone()` answers without waiting.

Step 5 / 10

VARIABLES

`f1.isDone() = false`

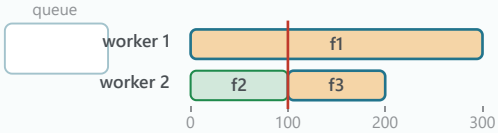
OUTPUT

```
false false
```

Times are approximate (ignoring thread start-up delay); which worker gets which task may vary from run to run.

The Pool and `Future`, Step by Step

```
1  static int job(int ms, int d) throws Exception {
2      Thread.sleep(ms);   // stands in for slow work
3      return d;
4  }
5  // inside main:
6  ExecutorService pool = Executors.newFixedThreadPool(2);
7  Future<Integer> f1 = pool.submit(() -> job(300, 10));
8  Future<Integer> f2 = pool.submit(() -> job(100, 20));
9  Future<Integer> f3 = pool.submit(() -> job(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
▶ int sum = f1.get() + f2.get() + f3.get();
12 System.out.println("sum = " + sum);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 pool.shutdown();
```



main: waiting in f1.get()

done: f2

`f1.get()` **blocks** `main` until the result is ready. Meanwhile f2 ends and worker 2 takes f3 from the queue.

Step 6 / 10

VARIABLES

`time = ≈100 ms`

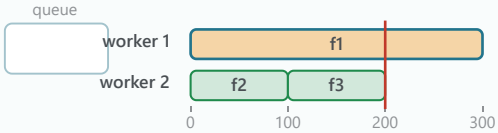
OUTPUT

```
false false
```

Times are approximate (ignoring thread start-up delay); which worker gets which task may vary from run to run.

The Pool and `Future`, Step by Step

```
 1 static int job(int ms, int d) throws Exception {
 2     Thread.sleep(ms);   // stands in for slow work
 3     return d;
 4 }
 5 // inside main:
 6 ExecutorService pool = Executors.newFixedThreadPool(2);
 7 Future<Integer> f1 = pool.submit(() -> job(300, 10));
 8 Future<Integer> f2 = pool.submit(() -> job(100, 20));
 9 Future<Integer> f3 = pool.submit(() -> job(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
▶ int sum = f1.get() + f2.get() + f3.get();
12 System.out.println("sum = " + sum);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 pool.shutdown();
```



main: waiting in f1.get()

done: f2, f3

f3 ends too; `main` is still waiting for f1.

Step 7 / 10

VARIABLES

`time = ≈200 ms`

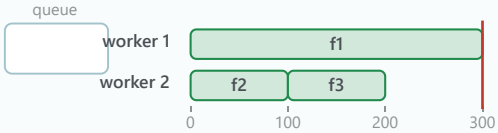
OUTPUT

```
false false
```

Times are approximate (ignoring thread start-up delay); which worker gets which task may vary from run to run.

The Pool and `Future`, Step by Step

```
1  static int job(int ms, int d) throws Exception {
2      Thread.sleep(ms);   // stands in for slow work
3      return d;
4  }
5  // inside main:
6  ExecutorService pool = Executors.newFixedThreadPool(2);
7  Future<Integer> f1 = pool.submit(() -> job(300, 10));
8  Future<Integer> f2 = pool.submit(() -> job(100, 20));
9  Future<Integer> f3 = pool.submit(() -> job(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
▶ int sum = f1.get() + f2.get() + f3.get();
12 System.out.println("sum = " + sum);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 pool.shutdown();
```



main: running

done: f1, f2, f3

t ≈ 300 ms: f1 is ready. f2 and f3 are already done, so their `get()` s return at once. Total time ≈ 300 ms, not 500.

Step 8 / 10

VARIABLES

`sum = 10+20+30=60`

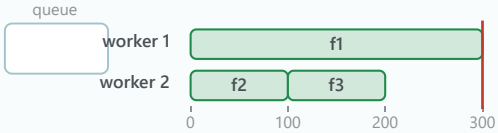
OUTPUT

```
false false
```

Times are approximate (ignoring thread start-up delay); which worker gets which task may vary from run to run.

The Pool and `Future`, Step by Step

```
1  static int job(int ms, int d) throws Exception {
2      Thread.sleep(ms);   // stands in for slow work
3      return d;
4  }
5  // inside main:
6  ExecutorService pool = Executors.newFixedThreadPool(2);
7  Future<Integer> f1 = pool.submit(() -> job(300, 10));
8  Future<Integer> f2 = pool.submit(() -> job(100, 20));
9  Future<Integer> f3 = pool.submit(() -> job(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int sum = f1.get() + f2.get() + f3.get();
▶  System.out.println("sum = " + sum);
▶  System.out.println(f2.isDone() + " " + f3.isDone());
14 pool.shutdown();
```



main: running

done: f1, f2, f3

The result and states are printed.

Step 9 / 10

VARIABLES

`sum = 60`

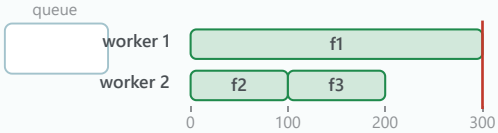
OUTPUT

```
false false
sum = 60
true true
```

Times are approximate (ignoring thread start-up delay); which worker gets which task may vary from run to run.

The Pool and `Future`, Step by Step

```
1  static int job(int ms, int d) throws Exception {
2      Thread.sleep(ms);   // stands in for slow work
3      return d;
4  }
5  // inside main:
6  ExecutorService pool = Executors.newFixedThreadPool(2);
7  Future<Integer> f1 = pool.submit(() -> job(300, 10));
8  Future<Integer> f2 = pool.submit(() -> job(100, 20));
9  Future<Integer> f3 = pool.submit(() -> job(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int sum = f1.get() + f2.get() + f3.get();
12 System.out.println("sum = " + sum);
13 System.out.println(f2.isDone() + " " + f3.isDone());
▶  pool.shutdown();
```



main: done

done: f1, f2, f3

`shutdown()` stops accepting tasks; the workers end. Forget it and the program never exits.

Step 10 / 10

VARIABLES

`pool = shutting down`

OUTPUT

```
false false
sum = 60
true true
```

Times are approximate (ignoring thread start-up delay); which worker gets which task may vary from run to run.

`CompletableFuture`, Concurrent Collections, `ForkJoinPool`

```
CompletableFuture<Integer> cf = CompletableFuture
    .supplyAsync(() -> 6)
    .thenApply(x -> x * 7)              // 42
    .thenCombine(CompletableFuture
        .supplyAsync(() -> 8), Integer::sum);
System.out.println(cf.get());           // 50

Map<String, Integer> freq =
    new ConcurrentHashMap<>();
String[] w = {"apple", "pear", "apple"};
Arrays.stream(w).parallel()
    .forEach(s -> freq.merge(s, 1, Integer::sum));
System.out.println(freq);  // {apple=2, pear=1}
```

- `CompletableFuture`: rather than blocking on `get()`, **chain** steps; each runs in the background after the previous one.
- `ConcurrentHashMap`: a safe map with fine-grained locking; `merge` and `compute` are atomic.
- `CopyOnWriteArrayList`: copies on every write; for read-mostly lists.
- `ForkJoinPool`: divide-and-conquer tasks (`RecursiveTask`); an idle worker **steals** from another's queue. Parallel streams (`parallelStream()`) use it.
- A parallel stream isn't always faster: small data and shared mutable state slow or break it; measure first.

OTHER TOOLS

Thread Safety

A class is **thread-safe** if it behaves correctly even when used from several threads at once.

Immutable objects

Objects that never change after construction (`String` , classes with `final` fields, Java 16+ `record` s) can be shared without synchronization.

Don't share

Local variables live on each thread's stack; `ThreadLocal` gives copies too. Unshared data can't race.

Synchronization

Every access to shared mutable data is guarded by the same lock: `synchronized` , `Lock` .

Ready-made structures

`AtomicInteger` , `ConcurrentHashMap` , `CopyOnWriteArrayList` , `BlockingQueue` ; `ArrayList` and `HashMap` are not safe.

Summary

Topic	Essence	Tools
Thread	An independent flow inside a process; shared heap, separate stack	`Thread`, `Runnable`
Starting	Work goes in `run()`; `start()` starts it	`start()`, `join()`
Life cycle	NEW → RUNNABLE ⇄ BLOCKED / WAITING / TIMED_WAITING → TERMINATED	`getState()`
Stopping	By cooperation: a flag or an interrupt	`volatile`, `interrupt()`
Race condition	Divisible operations on shared data; the result depends on order	`synchronized`, `Lock`
Visibility	What one thread writes may stay invisible to another	`volatile`
Deadlock	Taking locks in different orders creates a cycle	A fixed order, `tryLock`
Condition waiting	Wait while releasing the lock; wake on change	`wait/notify`, `Condition`
Coordination	Counting, meeting, swapping, permits	`CountDownLatch`, `CyclicBarrier`, `Exchanger`, `Semaphore`
Other	Lock-free counters, per-thread values, pools	`AtomicInteger`, `ThreadLocal`, `ExecutorService`

REVIEW

Common Mistakes (1)

Mistake	Symptom	Fix
Calling `t.run()`	No new thread; the code runs in order on main	`t.start()`
Calling `start()` twice	`IllegalThreadStateException`	A new `Thread` for each run
`Thread.sleep(100);` on its own	`error: unreported exception InterruptedException; must be caught or declared to be thrown`	`try-catch` or `throws InterruptedException`
Loop variable `i` in a lambda	`error: local variables referenced from a lambda expression must be final or effectively final`	A copy: `int no = i;`
Catching an exception that is never thrown: `try { while (!Thread.interrupted()) { } } catch (InterruptedException e)`	`error: exception InterruptedException is never thrown in body of corresponding try statement`	The body needs a `sleep` / `wait`, or drop the `catch`
`currentThread()` in a class that doesn't extend `Thread`	`error: cannot find symbol`	`Thread.currentThread()`
Forgetting `join()`	main reads the result before it's ready (e.g. 0)	`join()` before reading the result

REVIEW

Common Mistakes (2)

Mistake	Symptom	Fix
Calling `wait()` / `notify()` outside `synchronized`	`IllegalMonitorStateException: current thread is not owner`	Inside `synchronized` on the same object
Waiting on a condition with `if`	The condition may be false again on waking; an overflowing/empty queue	`while (condition) wait();`
`unlock()` outside `finally`	On an exception the lock is never released; everyone waits forever	`lock(); try { … } finally { unlock(); }`
Unlocking a lock you don't hold	`IllegalMonitorStateException`	One `unlock()` per `lock()`
`counter++` on a `volatile` counter	Increments get lost (e.g. 163299)	`AtomicInteger` or `synchronized`
A `final volatile` field	`error: illegal combination of modifiers: final and volatile`	Use only one of them
Using `stop()`, `suspend()`	Java 20+: `UnsupportedOperationException`	A flag, `interrupt()`, `wait/notify`
Swallowing `InterruptedException` in an empty `catch`	The interrupt request is lost; the thread doesn't stop	Leave the loop, or call `Thread.currentThread().interrupt()`
Taking locks in different orders	Deadlock: the program freezes	Everyone takes them in the same order

REVIEW

Practice Problems (1)

1) WHAT IS THE OUTPUT? IS IT THE SAME EVERY RUN?

```
Thread t = new Thread(() -> System.out.print("B"));
System.out.print("A");
t.start();
t.join();
System.out.print("C");
```

2) WHAT IS THE OUTPUT?

```
Thread t = new Thread(() -> System.out.print("X"));
System.out.print(t.getState() + " ");
t.run();
System.out.print(" " + t.getState());
```

3) WHICH OF THESE ARE POSSIBLE OUTPUTS OF THE "TWO THREADS" EXAMPLE?

a) A1 A2 B1 B2 main b) B1 A1 A2 B2 main
c) A1 main B1 A2 B2 d) A2 A1 B1 B2 main

▶ **Answers**

REVIEW

Practice Problems (2)

4) WHAT IS THE THREAD'S STATE IN EACH CASE?

a) inside `sleep(500)` · b) entering a `synchronized` block someone else holds · c) after `wait()` · d) when `run()` ends

5) IS THIS THREAD-SAFE?

```
class Account {
    private int balance = 0;
    void deposit(int m) { balance += m; }
    int getBalance() { return balance; }
}
```

6) TWO THREADS RUN THESE. WHAT CAN HAPPEN?

```
// T1
synchronized (a) {
  synchronized (b) { }
}
```

```
// T2
synchronized (b) {
  synchronized (a) { }
}
```

▶ **Answers**

REVIEW

Exercises (1)

- **1.** Sum the numbers 1..1000 with two threads: one adds the first half, the other the second; main waits for both and prints the total.
- **2.** Count the even numbers in 1..1000 with four threads (250 numbers each) into a shared counter. The count must come out right.

▶ **Answers**

REVIEW

Exercises (2)

- **3.** Compute the squares of 1..5 with `Callable` tasks in a 3-thread pool and add them up from the `Future` s.
- **4.** Write a thread that prints “tick” every second; main stops it with `interrupt()` after 3.5 s.

▶ **Answers**

References

- Oracle, *The Java Tutorials*: Lesson: Concurrency (Processes and Threads, Synchronization, Liveness, Guarded Blocks, High Level Concurrency Objects).
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition, Chapter 17: Threads and Locks.
- Oracle, *Java SE 21 API Documentation*: `java.lang.Thread`, `java.util.concurrent`, `java.util.concurrent.locks`, `java.util.concurrent.atomic`.
- B. Goetz et al., *Java Concurrency in Practice*, Addison-Wesley, 2006.
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 23: Concurrency).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapter 12: Concurrency).
- K. Sierra, B. Bates and T. Gee, *Head First Java*, 3rd ed., O'Reilly (the chapters on threads and concurrency).
- J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley (Chapter 11: Concurrency, Items 78–84).

The End

Chapter 10: Threads

Previous: Chapter 9: Graphics and User Interfaces · Next: Chapter 11: Database Programming