Threads

Java'da thread'ler; oluşturma (Thread/Runnable), lifecycle, thread metotları, race condition, synchronized, volatile, wait/notify, Atomic sınıflar, ReentrantLock ve deadlock.

İleri 55 dk

Son güncelleme: 10 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/threads

Threads

Buraya kadarki yedi konu (Enum'dan Polymorphism'e) hep tek bir iş parçacığında (thread) çalışan kodu ele aldı — programın her an yalnızca bir yerde olduğunu varsaydık. Bu konuyla birlikte **Concurrency** adlı yeni bir kategoriye giriyoruz: aynı anda birden fazla iş parçacığının çalıştığı, durumu paylaştığı ve bazen birbirine çarptığı bir dünya. Bu ilk Concurrency dersi, `Thread` sınıfının temel mekaniğine — oluşturma, yaşam döngüsü, senkronizasyon, `volatile`, kilitler ve deadlock'a — odaklanıyor; `ExecutorService`, `CompletableFuture` ve modern concurrency araçları ayrı bir sonraki derste ele alınacak.

Konu Nedir?

Bir **thread** (iş parçacığı), bir programın bağımsız olarak çalıştırılabilen en küçük yürütme birimidir. Bir **process** (süreç) — örneğin çalışan bir JVM örneği — kendi belleğine sahipken, o process içindeki thread'ler **aynı belleği paylaşır**; bu paylaşım hem thread'lerin gücünün hem de bu dersin ele alacağı çoğu problemin (race condition, deadlock) kaynağıdır. Her Java programı en az bir thread'le başlar — `main` metodunu çalıştıran **main thread**:

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

Bir programın birden fazla thread'i olduğunda, bunlara **multithreading** (çok iş parçacıklılık) denir — aynı process içinde birden fazla görev **eş zamanlı** (concurrent) ya da (çok çekirdekli bir işlemcide) gerçekten **paralel** (parallel) çalışabilir.

Neden Var?

Gerçek hayattan bir örnek: bir masaüstü uygulaması düşün — kullanıcı bir dosya indirirken arayüzün donmamasını, aynı anda fare tıklamalarına yanıt vermeye devam etmesini istersin. Tek thread'li bir programda, dosya indirme bitene kadar **hiçbir şey** çalışamaz — arayüz donar. Multithreading, indirme işini ayrı bir thread'e taşıyarak main thread'in (ve dolayısıyla arayüzün) serbest kalmasını sağlar. Aynı fikir sunucu tarafında da geçerli: bir web sunucusu, her gelen isteği ayrı bir thread'de işleyerek yüzlerce kullanıcıya **aynı anda** hizmet verebilir — sıraya sokup birer birer işlemek yerine.

Tarihçe

Thread kavramı işletim sistemleri dünyasında Java'dan çok daha eski, ama Java'yı diğer birçok dilden ayıran şey, thread desteğini **dilin ve standart kütüphanenin bir parçası** olarak, JDK 1.0'dan (1996) beri sunmasıydı — `Thread` sınıfı ve `synchronized` anahtar kelimesi dilin ilk gününden beri var. Bu erken karar, Java'yı o dönem sunucu tarafı yazılımlar için cazip kıldı. Ama ilk API'ler (`wait` / `notify` , düşük seviyeli `synchronized`) kullanımı zor ve hataya açıktı; Java 5 (2004), `java.util.concurrent` paketiyle (`ExecutorService` , `ConcurrentHashMap` , Atomic sınıflar gibi) çok daha yüksek seviyeli, güvenli araçlar getirdi — bir sonraki Concurrency dersinin ana konusu bu paket olacak. En son, Java 21 (2023), **virtual thread**'lerle (Project Loom) thread'lerin maliyetini kökten değiştirdi — bu, ileride ayrı bir "Modern Concurrency" dersinde ele alınacak kadar önemli bir gelişme.

Thread Oluşturma: Thread Sınıfını Extend Etmek

Bir thread oluşturmanın iki klasik yolundan ilki, `Thread` sınıfını extend edip `run()` metodunu override etmektir — Inheritance dersinin "Method Overriding" bölümünde gördüğümüz mekanizmanın doğrudan bir uygulaması:

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

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

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

`run()` içine yazdığın kod, `start()` çağrıldığında **yeni bir thread üzerinde** çalışır. Çıktının sırası (main mi yoksa yeni thread mi önce yazdırır) garanti değildir — işletim sisteminin zamanlayıcısına bağlıdır; bu öngörülemezlik, "Race Condition" bölümünde göreceğimiz problemlerin de temel nedenidir. `start()` ile `run()`'ı doğrudan çağırmak arasındaki kritik fark, "Thread Metotları: start(), join(), sleep(), interrupt()" bölümünde detaylı işleyeceğimiz bir konu — şimdilik `start()`'ın yeni bir thread başlattığını, `run()`'ın ise sadece normal bir metot çağrısı olduğunu bil.

Thread Oluşturma: Runnable Implement Etmek

İkinci ve genelde tercih edilen yol, `Runnable` interface'ini implement edip onu bir `Thread`'e vermektir. Bunun `Thread`'i extend etmeye üstünlüğü, Inheritance dersinin "Inheritance vs Composition" bölümünde işlediğimiz fikirle birebir örtüşüyor: `Thread`'i extend eden bir sınıf, Java'nın tek kalıtım kısıtlaması yüzünden **başka hiçbir sınıfı extend edemez**; `Runnable` implement eden bir sınıf ise dilediği başka sınıfı da extend edebilir:

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

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

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

`Task`, `Runnable`'ı implement ediyor ama kendisi bir `Thread` **değil** — `new Thread(task)` ile bir `Thread` nesnesine **verilip** çalıştırılıyor, tıpkı Inheritance dersindeki composition örneğinde `Engine`'in bir `Car`'a verilmesi gibi. Bu ayrım aynı zamanda "is-a" ile "has-a" arasındaki farkı da netleştiriyor: bir `Task` bir `Thread` değildir, yalnızca bir thread'in **çalıştıracağı iş**tir.

> Modern Java'da `Runnable` implementasyonları genellikle bir lambda ile yazılır: `new Thread(() -> System.out.println("çalışıyor")).start();` — `Runnable`'ın tek soyut metodu (`run()`) olduğu için bu, bir **functional interface**'tir (Interface dersinin "Functional Interface ve Lambda" bölümünü hatırla).

Thread Lifecycle

Bir thread, yaşamı boyunca `Thread.State` enum'unda tanımlı altı durumdan birinde bulunur (Enum dersinde gördüğümüz gibi, `Thread.State` de sabit bir değer kümesidir):

- **NEW:** `Thread` nesnesi oluşturuldu ama `start()` henüz çağrılmadı.
- **RUNNABLE:** `start()` çağrıldı, thread çalışıyor ya da CPU'ya sıra bekliyor.
- **BLOCKED:** Thread, başka bir thread'in elinde tuttuğu bir kilidi (lock) bekliyor.
- **WAITING:** Thread, `wait()` ya da `join()` gibi bir çağrıyla süresiz bekliyor.
- **TIMED_WAITING:** Thread, `sleep(ms)` ya da zaman aşımlı `wait(ms)` ile belirli bir süre bekliyor.
- **TERMINATED:** `run()` metodu tamamlandı, thread sona erdi.

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

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

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

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

`worker.getState()`'in üç farklı anda üç farklı değer döndürdüğüne dikkat et: `start()` çağrılmadan önce `NEW`, uyurken `TIMED_WAITING`, `join()` döndükten sonra `TERMINATED`. Bu örnekte `main`'in `Thread.sleep(50)` ile kısa bir süre beklemesi, worker'ın gerçekten uyku durumuna geçmiş olmasını garantilemek için — pratikte tam zamanlama işletim sistemine bağlı olsa da, 50ms'lik bekleme 200ms'lik bir uykuyu yakalamak için fazlasıyla yeterli.

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

Dört temel thread metodu var: `start()` yeni bir thread başlatır (bir kez daha: `run()`'ı doğrudan çağırmak yeni thread açmaz, sadece normal bir metot çağrısıdır); `join()`, çağıran thread'i (genelde main) hedef thread bitene kadar bekletir; `sleep(ms)`, çalışan thread'i belirtilen süre kadar duraklatır (`static` bir metottur, her zaman **çağıran** thread'i uyutur); `interrupt()`, bir thread'e "iptal isteği" gönderir — thread `sleep()` / `wait()` / `join()` gibi bloklayan bir çağrıda beklerken bu istek bir `InterruptedException` olarak ona ulaşır.

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

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

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

`worker.join()` çağrısı, main thread'in `worker` tamamen bitene kadar `"worker is done"` satırını yazdırmamasını **garanti eder** — `join()` olmasaydı bu sıralama şansa kalırdı. `another.interrupt()`, `another`'ı `sleep(1000)` içindeyken uyandırıyor ve `InterruptedException` fırlatıyor; bu, uzun süren bir işlemi dışarıdan iptal etmenin standart yolu.

> `catch (InterruptedException e)` bloğunda hiçbir şey yapmadan yutmak (`{}`) ciddi bir hata sayılır — interrupt sinyali kaybolur ve kodu çağıran hiç kimse iptal isteğinin gerçekleştiğini bilemez. Interrupt'ı ya yeniden fırlat ya da (örnekteki gibi) `Thread.currentThread().interrupt()` ile durumu geri yükle.

Daemon Thread'ler

Java, bir thread'i **daemon** olarak işaretleyebilir — normal (non-daemon, ya da "user") thread'lerden farkı, JVM'in çalışmaya devam edip etmeyeceğine karar verirken daemon thread'leri **saymaması**dır: tüm user thread'ler bittiğinde, hâlâ çalışan daemon thread'ler olsa bile JVM kapanır:

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

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

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

`backgroundLogger` bir daemon thread olarak işaretlendiği için, `main` sona erdiğinde (yalnızca 500ms sonra) JVM hemen kapanıyor — `backgroundLogger`'ın sonsuz döngüsü hiç tamamlanmasa bile. `setDaemon(true)` çağrısının `start()`'tan **önce** yapılması zorunludur; bir thread çalışmaya başladıktan sonra daemon durumu değiştirilemez.

> Garbage collector thread'i, JDK'nın kendi kullandığı klasik bir daemon thread örneğidir — uygulamanın "asıl işi" bitince onun da yarım kalmış bir iş yüzünden JVM'i canlı tutmasını istemezsin.

Race Condition

Şimdi asıl problem: iki thread aynı **paylaşılan durumu** (shared state) aynı anda değiştirmeye çalıştığında, sonuç beklenmedik ve **tekrarlanamaz** şekilde yanlış çıkabilir. Buna **race condition** denir:

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

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

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

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

`counter++` tek bir CPU işlemi gibi görünse de aslında üç ayrı adımdır: değeri oku, bir artır, geri yaz. İki thread bu üç adımı iç içe geçirirse (interleave), bir thread'in artırdığı değer diğerinin **eski okumasının üzerine yazılarak** kaybolabilir. Her iki thread de sayacı 100.000 kez artırsa bile, sonuç neredeyse hiçbir zaman beklenen 200.000 çıkmaz — ve her çalıştırmada **farklı** bir yanlış sayı görebilirsin, çünkü hangi thread'in ne zaman araya gireceği işletim sistemi zamanlayıcısına bağlı. Bu problemi "Synchronization" bölümünde çözeceğiz.

Synchronization

`synchronized` anahtar kelimesi, bir bölgeye aynı anda yalnızca **bir** thread'in girebilmesini garanti ederek race condition'ı çözer. Her Java nesnesinin görünmez bir kilidi (**intrinsic lock** ya da **monitor**) vardır; `synchronized` bir metoda ya da bloğa giren thread bu kilidi alır, çıkarken bırakır — kilit tutulduğu sürece başka hiçbir thread aynı kilide ihtiyaç duyan bir bölgeye giremez:

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

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

    int getCount() {
        return count;
    }
}

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

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

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

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

`increment()` metodu `synchronized` işaretlendiği için, bir thread bu metodun **içindeyken** başka hiçbir thread aynı `SafeCounter` nesnesinin `increment()`'ine giremez — "Race Condition" bölümündeki üç adımlık okuma-artırma-yazma artık bölünemez (atomik) hale geliyor. Sonuç bu sefer her zaman tam olarak beklenen 2000.

> `synchronized` bir blok (`synchronized (lockObject) { ... }`), `synchronized` bir metottan daha **dar kapsamlı** bir kilitleme sağlar — yalnızca gerçekten paylaşılan durumu değiştiren satırları kilitlemen, metodun geri kalanını (kilit gerektirmeyen kısımları) serbest bırakman anlamına gelir. Daha az kod kilitli kaldığı için genelde daha iyi performans verir.

volatile Anahtar Kelimesi

`volatile`, `synchronized`'la sıkça karıştırılır ama tamamen farklı bir problemi çözer: **memory visibility** (bellek görünürlüğü). Performans için her thread, paylaşılan bir değişkenin kendi

CPU çekirdeğine özel bir kopyasını (cache) tutabilir — bu yüzden bir thread bir değeri değiştirdiğinde, diğer thread'ler bu değişikliği **hiç görmeyebilir**. `volatile`, bir değişkenin her okuma/yazmasının doğrudan ana bellekten yapılmasını garanti eder:

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

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

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

`running` alanı `volatile` olmasaydı, JVM'in derleyici optimizasyonları `worker` thread'inin `running`'i yalnızca bir kez okuyup değerini sonsuza dek doğru sanmasına (ve döngüden asla çıkmamasına) yol açabilirdi — main'in `running = false` yazması, worker'ın gördüğü kopyaya hiç yansımayabilirdi. `volatile` bu görünürlüğü garanti ediyor.

> `volatile`, yalnızca **görünürlüğü** garanti eder, **atomikliği** garanti etmez. `volatile int counter` üzerinde `counter++` yazmak hâlâ bir race condition'dır — "Race Condition" bölümündeki üç adımlı problem aynen geçerlidir. Atomik artırma için `synchronized` ya da "Atomic Sınıflar" bölümünde göreceğimiz `AtomicInteger` gerekir.

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

`synchronized`, thread'lerin birbirini **engellemesini** sağlar; `wait()` / `notify()` / `notifyAll()` ise thread'lerin birbirine **haber vermesini** sağlar — bir thread belirli bir koşul gerçekleşene kadar beklemesi gerektiğinde (örneğin bir kuyruk boşken tüketmemesi gerektiğinde), `wait()` ile kilidi geçici olarak bırakıp uyur; koşulu değiştiren başka bir thread `notify()` / `notifyAll()` ile onu uyandırır:

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

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

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

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

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

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

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

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

`wait()` ve `notify()`'ın **her ikisi de** `synchronized` bir blok/metot içinde çağrılmak zorundadır — aksi halde `IllegalMonitorStateException` fırlatılır, çünkü ikisi de aynı nesnenin intrinsic lock'una (monitor) ihtiyaç duyar. `wait()`'in bir `while` döngüsü içinde (asla tek bir

`if` ile değil) çağrılması da kritik: thread uyandığında koşulun **hâlâ** geçerli olduğunu tekrar kontrol etmen gerekir — "spurious wakeup" denen, hiçbir `notify()` çağrılmadan thread'in uyanabilmesi ihtimaline karşı.

Modern Java kod tabanlarında `wait()` / `notify()` yerine genelde `java.util.concurrent` paketindeki daha yüksek seviyeli araçlar (`BlockingQueue`, `CountDownLatch` gibi) tercih edilir — bunlar aynı fikri, spurious wakeup ve yanlış kilit kullanımı gibi tuzaklara düşmeden sağlar; bir sonraki Concurrency dersinde bunlara değineceğiz.

Atomic Sınıflar

`AtomicInteger`, `AtomicLong` ve `AtomicReference` gibi sınıflar, `java.util.concurrent. atomic` paketinde, `synchronized` kullanmadan atomik (bölünemez) işlemler sağlar. İçlerinde donanım seviyesinde bir **CAS** (compare-and-swap) işlemi kullanırlar: "bu değişkenin değeri hâlâ X ise, Y yap; değilse tekrar dene" — kilit almadan çalışan, çok daha hafif bir mekanizma:

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

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

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

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

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

`AtomicInteger.incrementAndGet()`, "Race Condition" bölümündeki `counter++`'ın atomik karşılığı — okuma, artırma ve yazma tek bir bölünemez işlem olarak gerçekleşir, `synchronized` bloğuna hiç gerek kalmadan. Basit sayaçlar, bayraklar ya da referans güncellemeleri için Atomic sınıflar genelde `synchronized`'dan daha hafif ve daha hızlıdır — ama karmaşık, birden fazla adımlı işlemler için (örneğin bir sonraki mini projedeki banka hesabı gibi birden fazla kararı tutarlı tutman gerektiğinde) `synchronized` ya da bir `Lock` genelde daha uygun bir araçtır.

Locks: ReentrantLock

`java.util.concurrent.locks` paketindeki `ReentrantLock`, `synchronized`'ın yapabildiği her şeyi yapabilen, ama daha fazla kontrol sunan bir kilitleme aracıdır: kilidi almayı **denemek** (`tryLock()`) ya da belirli bir süre beklemek gibi `synchronized`'ın sunmadığı esneklikler sağlar:

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

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

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

    int getCount() {
        return count;
    }
}

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

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

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

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

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

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

`lock()` ile `unlock()` arasındaki kod, tıpkı `synchronized` bloğu gibi aynı anda yalnızca bir thread tarafından çalıştırılabilir — ama `unlock()`'un mutlaka bir `finally` bloğunda çağrılması **gerekir**, aksi halde aradaki kod bir exception fırlatırsa kilit sonsuza dek elde tutulur (`synchronized` bunu otomatik garanti eder, `ReentrantLock` etmez). `tryLock()`, bir thread'in kilidi bekleyerek sonsuza dek bloke olmak yerine, kilit meşgulse **hemen vazgeçip** başka bir iş yapmasına izin veriyor — `synchronized` ile bu mümkün değil.

> `ReentrantLock` kullanırken `unlock()`'u `finally` bloğunun dışında ya da hiç çağırmamak, en sık yapılan hatadır — kilit asla serbest bırakılmaz ve o kilidi bekleyen tüm thread'ler sonsuza dek bloke kalır. Bu, "Deadlock" bölümünde göreceğimiz sorunlardan biriyle aynı sonucu doğurur.

Deadlock

Bir **deadlock**, iki (ya da daha fazla) thread'in birbirinin elinde tuttuğu bir kilidi beklerken **sonsuza dek** birbirini bloke etmesidir. En klasik senaryo: `Thread A` `lockA`'yı tutup `lockB`'yi beklerken, `Thread B` aynı anda `lockB`'yi tutup `lockA`'yı bekliyorsa, ikisi de asla ilerleyemez:

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

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

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

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

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

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

`t1`, önce `lockA`'yı kilitleyip `lockB`'yi beklerken; `t2` önce `lockB`'yi kilitleyip `lockA`'yı bekliyor — iki thread de birbirinin tuttuğu kilidi istiyor ve hiçbiri asla vazgeçmiyor. Bunu

önlemenin en yaygın yolu, **tüm thread'lerin kilitleri her zaman aynı sırada almasını** garanti etmek — örneğin her zaman önce "küçük" olan kilidi almak gibi; bu, "Best Practices" bölümünde tekrar değineceğimiz bir kural.

> Gerçek bir deadlock'ta program sonsuza dek asılı kalır — bu örnekte, yalnızca öğretim amacıyla, bir "gözcü" (`join` zaman aşımı) ekleyerek deadlock'u **tespit edip raporluyoruz**, gerçek hayatta böyle bir güvenlik ağı yok. Gerçek bir uygulamada asla iki kilidi bu şekilde farklı sırayla almaya çalışma.

Best Practices

- Race condition'dan korunmak için paylaşılan değişebilir durumu her zaman `synchronized`, bir `Lock` ya da bir Atomic sınıfla koru — hiçbirini kullanmadan "muhtemelen sorun olmaz" diye düşünme (bkz. "Race Condition").
- Kilitleri her zaman **aynı sırada** al — deadlock'ların çoğu, farklı thread'lerin farklı sırada kilit almasından kaynaklanır (bkz. "Deadlock").
- `wait()`'i her zaman bir `while` döngüsü içinde çağır, tek bir `if` ile değil — spurious wakeup'lara karşı korunmak için (bkz. "Thread Communication: wait(), notify(), notifyAll()").
- `ReentrantLock` kullanıyorsan `unlock()`'u mutlaka bir `finally` bloğunda çağır (bkz. "Locks: ReentrantLock" bölümündeki uyarı).
- Basit bir sayaç/bayrak için `synchronized` yerine önce Atomic sınıfları düşün — genelde daha hafif ve daha az hataya açıktır (bkz. "Atomic Sınıflar").
- Mümkünse paylaşılan değişebilir durumdan tamamen kaçın — değişmez (immutable) nesneler hiçbir zaman race condition'a konu olamaz, çünkü değiştirilemezler.

Yaygın Hatalar

1. `start()` yerine `run()`'ı çağırmak. `run()`'ı doğrudan çağırmak yeni bir thread açmaz — kod, çağıran thread üzerinde, normal bir metot çağrısı gibi çalışır (bkz. "Thread Oluşturma: Thread Sınıfını Extend Etmek").

2. `InterruptedException`'ı boş bir `catch` bloğuyla yutmak. Interrupt sinyali kaybolur ve kodu çağıran taraf iptalin gerçekleştiğini asla öğrenemez (bkz. "Thread Metotları: start(), join(), sleep(), interrupt()" bölümündeki uyarı).

3. `volatile`'ın atomiklik sağladığını sanmak. `volatile` yalnızca görünürlüğü garanti eder — `volatile` bir sayaç üzerinde `counter++` hâlâ bir race condition'dır (bkz. "volatile Anahtar Kelimesi" bölümündeki uyarı).

4. `wait()`'i bir `if` ile çağırıp `while` kullanmamak. Spurious wakeup ihtimaline karşı, uyanan thread'in koşulu **tekrar** kontrol etmesi gerekir (bkz. "Thread Communication: wait(), notify(), notifyAll()").

5. Kilitleri farklı thread'lerde farklı sırayla almak. Bu, deadlock'ların en yaygın nedenidir — tüm thread'lerin kilitleri aynı sırada aldığından emin ol (bkz. "Deadlock").

6. `ReentrantLock.unlock()`'u `finally` dışında çağırmak. Kilitli bölge bir exception fırlatırsa kilit asla serbest bırakılmaz (bkz. "Locks: ReentrantLock" bölümündeki uyarı).

Özet, Cheat Sheet ve Terimler Sözlüğü

Threads, Java'nın JDK 1.0'dan beri sahip olduğu, aynı process içinde birden fazla görevi eş zamanlı çalıştırma mekanizmasıdır. Öne çıkan noktalar:

- Bir thread oluşturmanın iki yolu var: `Thread`'i extend etmek ya da (genelde tercih edilen) `Runnable` implement etmek
- Bir thread altı durumdan birinde bulunur: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED
- `start()` yeni bir thread açar, `run()`'ı doğrudan çağırmak açmaz
- `join()` bir thread'in bitmesini bekler, `sleep()` çalışan thread'i duraklatır, `interrupt()` bloklayan bir çağrıyı `InterruptedException` ile keser
- Race condition, paylaşılan durumu koruma altına almadan birden fazla thread'in değiştirmesinden doğar
- `synchronized`, aynı anda yalnızca bir thread'in bir bölgeye girmesini garanti ederek race condition'ı çözer
- `volatile`, atomiklik değil yalnızca bellek görünürlüğü sağlar
- `wait()` / `notify()` / `notifyAll()`, `synchronized` içinde thread'lerin birbirine haber vermesini sağlar; `wait()` her zaman `while` içinde çağrılmalı
- Atomic sınıflar (`AtomicInteger` gibi), CAS ile kilitsiz atomik işlemler sağlar
- `ReentrantLock`, `synchronized`'a göre `tryLock()` gibi ek esneklikler sunar, ama `unlock()`'u elle ve `finally` içinde çağırmayı gerektirir
- Deadlock, thread'lerin birbirinin tuttuğu kilidi beklerken sonsuza dek bloke olmasıdır; kilitleri her zaman aynı sırada almak bunu önler

Hızlı referans:

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

// Temel metotlar
t1.join();          // bitmesini bekle
Thread.sleep(100);  // çalışan thread'i duraklat
t1.interrupt();     // bloklayan çağrıyı InterruptedException ile kes

// Race condition -> synchronized ile çözüm
class Counter {
    private int count = 0;
    synchronized void increment() { count++; } // tek seferde bir thread
}

// volatile -- yalnızca görünürlük
private volatile boolean running = true;

// Atomic -- kilitsiz atomik işlem
AtomicInteger counter = new AtomicInteger(0);
counter.incrementAndGet();

// ReentrantLock
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
    // ...
} finally {
    lock.unlock(); // mutlaka finally'de
}

// Deadlock'tan kaçınma: kilitleri her zaman aynı sırada al
synchronized (lockA) {
    synchronized (lockB) {
        // ...
    }
}
```

Terimler Sözlüğü

Thread (iş parçacığı) — Bir programın bağımsız çalıştırılabilen en küçük yürütme birimi; aynı process içindeki thread'ler belleği paylaşır.

Process (süreç) — Kendi belleğine sahip, çalışan bir program örneği (örneğin bir JVM örneği); içinde bir ya da daha fazla thread barındırır.

Race condition — Birden fazla thread'in paylaşılan durumu koruma altına almadan aynı anda değiştirmesinden doğan, tekrarlanamaz ve öngörülemez hata.

`synchronized` — Bir bölgeye aynı anda yalnızca bir thread'in girebilmesini, o nesnenin intrinsic lock'unu (monitor) kullanarak garanti eden anahtar kelime.

`volatile` — Bir değişkenin her okuma/yazmasının ana bellekten yapılmasını garanti eden, yalnızca görünürlük sağlayan (atomiklik sağlamayan) anahtar kelime.

Deadlock — İki ya da daha fazla thread'in, birbirinin tuttuğu bir kilidi beklerken sonsuza dek birbirini bloke etmesi.

Daemon thread — JVM'in kapanıp kapanmayacağına karar verirken sayılmayan, arka plan thread'i; tüm user thread'ler bittiğinde JVM, çalışan daemon thread'ler olsa bile kapanır.

CAS (compare-and-swap) — Atomic sınıfların, kilit almadan atomik güncelleme yapmak için kullandığı donanım seviyesi işlem.

`ReentrantLock` — `synchronized`'a göre `tryLock()`, zaman aşımlı bekleme gibi ek esneklikler sunan, `java.util.concurrent.locks` paketindeki açık kilitleme aracı.

Ek: Mini Proje — Thread-Safe Banka Hesabı

Bu mini projede, "Race Condition" bölümünde gördüğümüz problemi gerçekçi bir senaryoya taşıyoruz: birden fazla thread aynı banka hesabından **aynı anda** para çekmeye çalıştığında ne olur, ve bunu `synchronized` ile nasıl güvenli hale getiririz:

```
class UnsafeBankAccount {
    private int balance;

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

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

    int getBalance() {
        return balance;
    }
}

class SafeBankAccount {
    private int balance;

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

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

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

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

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

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

`UnsafeBankAccount.withdraw(...)` hiçbir koruma içermiyor — birden fazla thread aynı anda çekim yaptığında, "Race Condition" bölümündeki gibi bir güncelleme kaybolabiliyor ve hesap bakiyesi **matematiksel olarak imkansız** bir değere düşebiliyor (hatta negatife bile gidebiliyor). `SafeBankAccount` ise `withdraw(...)`'u `synchronized` yaparak aynı anda yalnızca bir thread'in bakiyeyi kontrol edip düşürmesine izin veriyor — bakiye kontrolü ile güncellemesi artık bölünemez tek bir işlem.

> Burada `synchronized` yerine `AtomicInteger` kullanmadık, çünkü çekim işlemi tek bir alanı değil, **hem bakiyeyi okuyup hem de yeterli olup olmadığını kontrol eden** birden fazla adımı içeriyor ("Atomic Sınıflar" bölümünün son cümlesini hatırla) — bu tür "kontrol et, sonra değiştir" işlemleri için `synchronized` / `Lock` genelde daha doğru araçtır.

Ek: Mini Proje — Producer/Consumer

Son mini proje, "Thread Communication: wait(), notify(), notifyAll()" bölümündeki fikri, klasik **Producer/Consumer** problemine genişletiyor: sınırlı kapasiteli paylaşılan bir kuyruk, producer dolu kuyruğa eklemeyi, consumer boş kuyruktan almayı beklemek zorunda:

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

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

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

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

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

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

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

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

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

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

`SharedQueue`, kapasitesi dolduğunda `put(...)`'u çağıran producer'ı `wait()` ile bekletiyor; kuyruk boşaldığında ise `take()`'i çağıran consumer'ı bekletiyor — her iki taraf da diğerinin `notifyAll()`'uyla uyanıyor. Bu, "Thread Communication: wait(), notify(), notifyAll()" bölümündeki fikrin, gerçek dünyada çok daha yaygın olan **sınırlı tampon** (bounded buffer) haline genişletilmiş hali.

> Gerçek bir uygulamada bu deseni sıfırdan `wait()` / `notify()` ile yazmak yerine, `java.util.concurrent.BlockingQueue`'nun hazır implementasyonlarını (`ArrayBlockingQueue` gibi) kullanırsın — bir sonraki Concurrency dersinde göreceğiz. Burada elle yazmamızın amacı, `BlockingQueue`'nun **içinde** aslında ne olduğunu anlamak.