

Bölüm 9: İş Parçacıkları

Java ile Nesne Yönelimli Programlama

İş parçacığı ve süreç

Thread ve Runnable

Yaşam döngüsü

sleep · join · daemon

Durdurma

Yarış durumu

synchronized

volatile

Lock

Kilitlenme

wait · notify · Condition

Senkronizasyon nesneleri

Atomik sınıflar

ThreadLocal

İş parçacığı havuzu

Öğrenme Çıktıları

Kavramlar

İş parçacığı ile süreç, eş zamanlılık ile paralellik arasındaki farkı açıklar.

Oluşturma

`Runnable` gerçekleştirerek ya da `Thread` sınıfını genişleterek iş parçacığı yazar, `start()` ile başlatır.

Yaşam döngüsü

Altı durumu ve `sleep`, `join`, `wait`, kilit beklemesi gibi geçişleri izler.

Senkronizasyon

Yarış durumunu tanır; `synchronized`, `volatile`, `Lock` ve atomik sınıflarla önler.

Koşul bekleme

`wait/notify` ve `Condition` ile üretici-tüketici yazar, kilitlemeyi önler.

Hazır araçlar

`CountDownLatch`, `CyclicBarrier`, `Exchanger`, `Semaphore`, `ThreadLocal` ve havuzları kullanır.

İş Parçacığı (Thread)

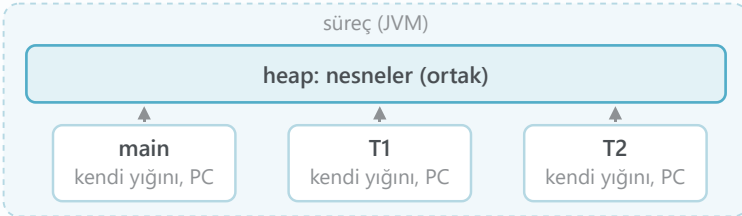
- İş parçacığı (thread), bir Java programındaki **temel yürütme birimidir**: komutları sırayla çalıştıran bağımsız bir akış.
- Her iş parçacığı, bir `Thread` sınıfı nesnesi tarafından temsil edilir.
- İş parçacıkları, **çoklu görev** (multithreading) ve **eş zamanlı programlama** (concurrent programming) için kullanılır.
- Her Java programı en az bir iş parçacığıyla başlar: `main()` metodunu çalıştıran **ana iş parçacığı** (main thread).

Benzetme: arı kovanı

Kovadaki her arı kendi işini yapar (bir iş parçacığı), ama hepsi aynı kovanı ve aynı bal peteklerini paylaşır (ortak bellek). İki arı aynı göze aynı anda bal koymaya çalışırsa sorun çıkar: bu bölümün konusu, bu "çarşışmaları" düzenlemektir.

Neden?

Arayüz donmadan arka planda iş yapmak, çok çekirdekli işlemciyi kullanmak, ağ ya da dosya beklerken başka işe devam etmek.



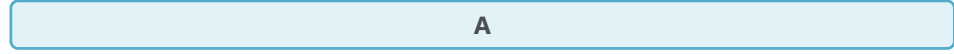
Eş Zamanlılık ve Paralellik

Tek çekirdek
eş zamanlılık

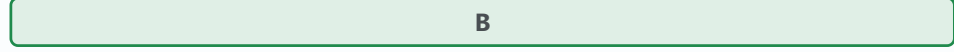


İki çekirdek
paralellik

Ç1



Ç2



zaman

Eş zamanlılık (concurrency)

Birden fazla görevin **aynı zaman aralığında ilerlemesi**. Tek çekirdekte bile olur: işletim sistemi iş parçacıklarını çok kısa dilimlerle sırayla çalıştırır, biz aynı anda çalışıyor sanırız.

Paralellik (parallelism)

Görevlerin **gerçekten aynı anda**, farklı çekirdeklerde çalışması. Birden fazla çekirdek gerektirir; hesaplamayı hızlandırır.

Hangi iş parçacığının ne zaman, hangi çekirdekte çalışacağına JVM ve işletim sisteminin **zamanlayıcısı** (scheduler) karar verir; programcı bu sırayı bilemez.

Süreç ve İş Parçacığı

Süreç: çalışan bir JVM

Heap: nesneler ve diziler
tüm iş parçacıkları ORTAK kullanır

main

Yığın (stack)
yerel değişkenler

PC: sıradaki komut

Thread-0

Yığın (stack)
yerel değişkenler

PC: sıradaki komut

	Süreç (process)	İş parçacığı
Bellek	Kendi adres alanı	Sürecin heap'ini paylaşır
Yığın	—	Her birinin kendi yığını
Oluşturma	Pahalı	Ucuz (hafif süreç)
İletişim	Dosya, soket, boru	Ortak nesneler
Risk	Birbirini bozamaz	Ortak veriyi bozabilir

Çalışan bir Java programı **bir süreçtir**; içinde birçok iş parçacığı çalışır. Nesneler heap'te olduğu için iki iş parçacığı **aynı nesneye** erişebilir; yerel değişkenler ise her iş parçacığının kendi yığınınındadır ve paylaşılmaz.

Bağlam Değiştirme (Context Switching)

- Çekirdek sayısından çok iş parçacığı varsa işletim sistemi onları **zaman dilimleriyle** sırayla çalıştırır.
- Geçişte çalışan iş parçacığının durumu (program sayacı, yazmaçlar, yığın işaretçisi) **saklanır**, sıradakinin durumu **yüklenir**: buna bağlam değiştirme denir.

tek çekirdek:



▲ bağlam değiştirme: kayıtları sakla, ötekini yükle

Her geçiş zaman alır; iş parçacığı sayısı çekirdek sayısını çok aşarsa bu maliyet büyür.

Maliyet

Her geçiş birkaç mikrosaniye sürer ve önbellekteki verinin bir kısmı geçersiz kalır. Binlerce iş parçacığı açmak programı **yavaşlatabilir**.

Kural

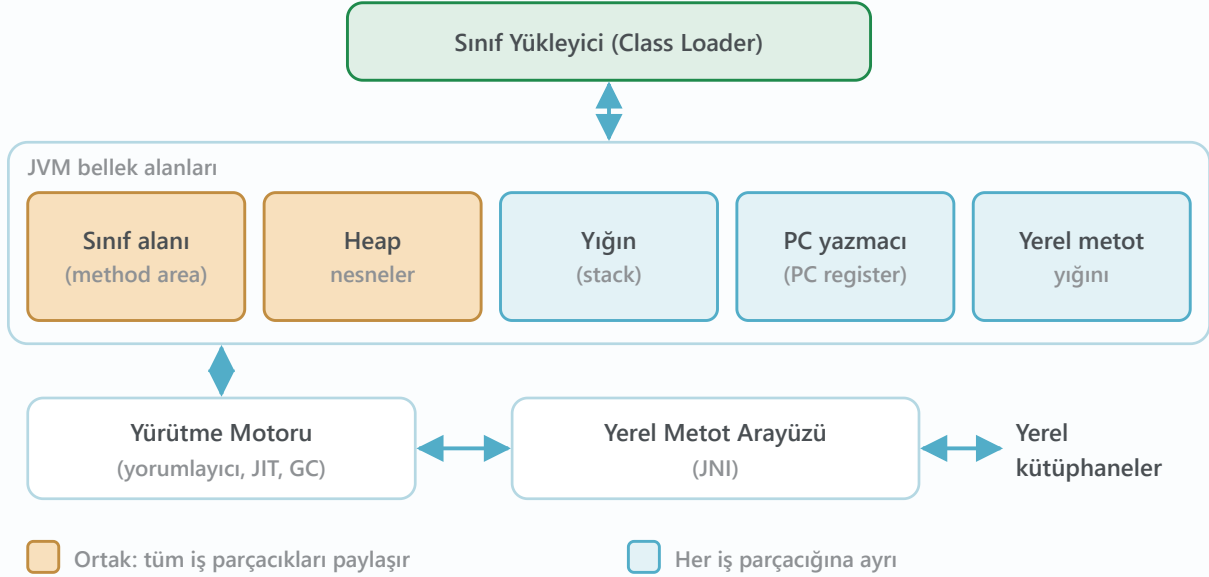
Hesap ağırlıklı işlerde iş parçacığı sayısı $\approx$ çekirdek sayısı

(`Runtime.getRuntime().availableProcessors()`).

Bekleme ağırlıklı (ağ, dosya) işlerde daha fazlası yararlı olabilir; bunun için havuz kullanılır.

Java Sanal Makinesi (JVM)

JVM, iş parçacıklarını **bağımsız yürütme birimleri** olarak yöneten sanal bir ortamdır.



Her iş parçacığı oluşturulduğunda JVM ona ayrı bir yığın, PC yazmacı ve yerel metot yığını ayırır; sınıf alanı ve heap tek kopyadır.

THREAD SINIFI

Thread Sınıfı

- Her iş parçacığı, `java.lang` paketindeki `Thread` sınıfının bir nesnesidir (import gerekmez).
- `Thread`, iş parçacığının **yaşam döngüsünü** yönetir; gerekli metotları ve özellikleri içerir.
- Bir yol: `Thread` 'den türeyen alt sınıf yazmak.
- `run()` metodu geçersiz kılınarak (override) iş parçacığının ana mantığı tanımlanır.
- `start()` çağrılarak iş parçacığı başlatılır.

Metot	Görevi
<code>start()</code>	Yeni iş parçacığını başlatır, o da <code>run()</code> 'ı çalıştırır
<code>run()</code>	İş parçacığının yapacağı iş
<code>sleep(ms)</code>	<i>static</i> : çalışanı belirli süre uyutur
<code>join()</code>	O iş parçacığının bitmesini bekler
<code>yield()</code>	<i>static</i> : yürütmeyi geçici olarak başkasına bırakır
<code>interrupt()</code>	Kesme isteği gönderir
<code>currentThread()</code>	<i>static</i> : şu an çalışan iş parçacığı
<code>getName()</code> , <code>getState()</code> , <code>isAlive()</code>	Bilgi okuma

İş Parçacığının Temel Nitelikleri (1)

Runnable (çalıştırılabilir)

`run()` metodunu içeren nesne. İş parçacığının ana mantığı burada tanımlanır.

Name (isim)

İş parçacığının adı: `main`,
`Thread-0` ... Günlük kayıtları (log) ve teşhis için kullanılır.
`getName()`, `setName()`

ID (kimlik)

Tekil kimlik numarası; iş parçacığı oluşturulurken sistem tarafından **otomatik** atanır.
`threadId()`

ThreadGroup (grup)

İş parçacığının ait olduğu grup; iş parçacıklarını düzenlemek ve yönetmek için kullanılır.

Daemon (arka plan)

Diğer iş parçacıklarına hizmet eder, belirli görevleri periyodik yapar; JVM kapanırken **bitmesi beklenmez**.

ContextClassLoader

Bağlam sınıf yükleyici: iş parçacığının sınıfları dinamik olarak yüklerken kullandığı sınıf yükleyici.

Bu nitelikler `Thread` nesnesinin alanlarıdır; okuma/yazma için get/set metotları vardır.

İş Parçacığının Temel Nitelikleri (2)

Priority (öncelik)

`MIN_PRIORITY` (1) ile
`MAX_PRIORITY` (10) arasında
tam sayı; varsayılan
`NORM_PRIORITY` (5).

State (durum)

Mevcut durum: `NEW` ,
`RUNNABLE` , `BLOCKED` ,
`WAITING` , `TIMED_WAITING` ,
`TERMINATED` .

Interrupted (kesilme)

Kesilme isteği gelip gelmediğini
gösteren bayrak;
`interrupt()` ile işaretlenir.

```
Thread t = Thread.currentThread(); // çalışan
System.out.println("Ad      : " + t.getName());
// threadId(): Java 19+ (öncesinde getId())
System.out.println("Kimlik   : " + t.threadId());
System.out.println("Öncelik  : " + t.getPriority());
String grup = t.getThreadGroup().getName();
System.out.println("Grup     : " + grup);
System.out.println("Daemon   : " + t.isDaemon());
System.out.println("Durum    : " + t.getState());
```

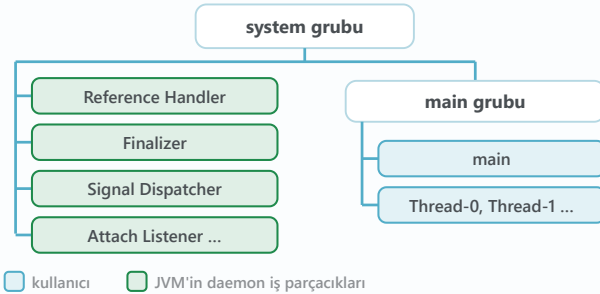
Çıktı

```
Ad      : main
Kimlik   : 1
Öncelik  : 5
Grup     : main
Daemon   : false
Durum    : RUNNABLE
```

Kod parçası `main` içindedir. `Thread.currentThread()` o satırı çalıştıran iş parçacığını verir; burada ana iş parçacığı.

THREAD SINIFI

JVM Başlangıcında İki İş Parçacığı Grubu



Main thread group

JVM başlarken oluşturulur; ana iş parçacığı ve bizim oluşturduğlarımız varsayılan olarak buraya aittir.

System thread group

JVM'in kendi iş parçacıkları: referans yönetimi, sonlandırıcı (Finalizer), sinyal gönderici (Signal Dispatcher)...

TÜM İŞ PARÇACIKLARINI LİSTELEYELİM

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

Çıktı (JDK 21, bir çalıştırma)

```
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
```

JVM Başlangıcında Dört Temel İş Parçacığı

Main Thread (ana iş parçacığı)

JVM başladığında ilk çalışan iş parçacığı.

`main()` metodu burada çalışır; temel uygulama mantığı burada yürütülür.

Garbage Collector (çöp toplayıcı)

Bellek yönetiminden sorumludur.

Referansı kalmamış nesneleri bulup temizler; **otomatik** çalışır. HotSpot'ta JVM'in iç iş parçacıklarıdır, listede görünmez.

Signal Dispatcher (sinyal gönderici)

İşletim sisteminden gelen sinyalleri yönetir; örneğin Ctrl+C ile gelen kapatma isteğinde JVM'in güvenli kapanmasını başlatır.

Finalizer (sonlandırıcı)

Nesnelerin `finalize()` metodunu çağırır. `finalize()` Java 9'da kullanımdan kaldırıldı (deprecated); yerine `try-finally` veya `AutoCloseable` + `try-with-resources` kullanılır.

YAŞAM DÖNGÜSÜ

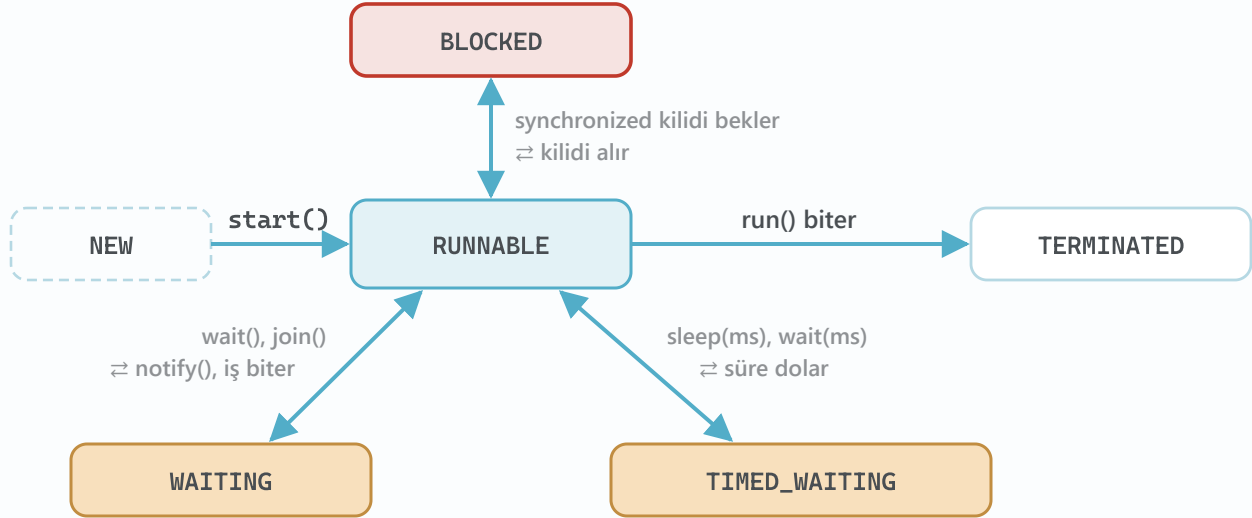
İş Parçacığı Durumları

İş parçacıkları yaşam döngüleri boyunca farklı durumlarda bulunur; `getState()` durumu `Thread.State` sabiti olarak verir.

Durum	Anlamı	Nasıl girilir?
<code>NEW</code> (yeni)	Nesne oluşturuldu, henüz başlatılmadı	<code>new Thread(...)</code> ; çıkış: <code>start()</code>
<code>RUNNABLE</code> (çalışabilir)	Çalışıyor ya da işlemci sırası bekliyor	<code>start()</code> sonrası
<code>BLOCKED</code> (engellenmiş)	<code>synchronized</code> blok/metoda girmek için monitör kilidini bekliyor	Kilit başkasındayken
<code>WAITING</code> (bekliyor)	Süresiz bekliyor; başka bir iş parçacığı uyandırmalı	<code>Object.wait()</code> , <code>Thread.join()</code> , <code>LockSupport.park()</code>
<code>TIMED_WAITING</code>	Sürelili bekliyor	<code>Thread.sleep(ms)</code> , <code>wait(ms)</code> , <code>join(ms)</code> , <code>parkNanos</code> , <code>parkUntil</code>
<code>TERMINATED</code> (sonlandı)	Yürütmeyi tamamladı	<code>run()</code> bitince

Dikkat: Java'da ayrı bir "RUNNING" durumu yoktur; çalışan ve sıra bekleyen iş parçacıklarının ikisi de `RUNNABLE` 'dır.

Durum Diyagramı



Tek yön

NEW → **RUNNABLE** → **TERMINATED**. Biten iş parçacığı yeniden başlatılamaz.

Bekleme durumları

Üç bekleme durumunun her birinden yeniden **RUNNABLE** 'a dönülür.

Fark

BLOCKED : kilit için;
WAITING : bir olay için;
TIMED_WAITING : süre sınırlı.

Durumları Gözlemlemek: Adım Adım

```
1 public static void main(String[] args) {  
2     Thread uyuyan = new Thread(() -> uyu(1000));  
3     System.out.println("1) " + uyuyan.getState());  
4     uyuyan.start();  
5     uyu(100);  
6     System.out.println("2) " + uyuyan.getState());  
7     Thread bekleyen = new Thread(() -> katil(uyuyan));  
8     bekleyen.start();  
9     uyu(100);  
10    System.out.println("3) " + bekleyen.getState());  
11    Thread engellenen = new Thread(() -> {  
12        synchronized (kilit) { }  
13    });  
14    synchronized (kilit) {  
15        engellenen.start();  
16        uyu(100);  
17        System.out.println("4) " + engellenen.getState());  
18    }  
19    katil(bekleyen);  
20    System.out.println("5) " + uyuyan.getState());  
21 }
```

main

RUNNABLE

uyuyan

NEW

bekleyen

henüz yok

engellenen

henüz yok

uyuyan nesnesi oluşturuldu ama başlatılmadı:

NEW

Adım 1 / 11

ÇIKTI

`uyu(ms)` ve `katil(t)` : `Thread.sleep` ve `t.join()` çağrılarını try-catch ile saran yardımcı metotlar. Bekleme süreleri sayesinde çıktı hep böyledir.

Durumları Gözlemlemek: Adım Adım

```
1 public static void main(String[] args) {  
2     Thread uyuyan = new Thread(() -> uyu(1000));  
3     System.out.println("1) " + uyuyan.getState());  
4     uyuyan.start();  
5     uyu(100);  
6     System.out.println("2) " + uyuyan.getState());  
7     Thread bekleyen = new Thread(() -> katil(uyuyan));  
8     bekleyen.start();  
9     uyu(100);  
10    System.out.println("3) " + bekleyen.getState());  
11    Thread engellenen = new Thread(() -> {  
12        synchronized (kilit) { }  
13    });  
14    synchronized (kilit) {  
15        engellenen.start();  
16        uyu(100);  
17        System.out.println("4) " + engellenen.getState());  
18    }  
19    katil(bekleyen);  
20    System.out.println("5) " + uyuyan.getState());  
21 }
```

main

RUNNABLE

uyuyan

NEW

bekleyen

henüz yok

engellenen

henüz yok

Durum yazdırılır.

Adım 2 / 11

ÇIKTI

1) NEW

`uyu(ms)` ve `katil(t)` : `Thread.sleep` ve `t.join()` çağrılarını try-catch ile saran yardımcı metotlar. Bekleme süreleri sayesinde çıktı hep böyledir.

Durumları Gözlemlemek: Adım Adım

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

main

RUNNABLE

uyuyan

RUNNABLE

bekleyen

henüz yok

engellenen

henüz yok

start() : yeni iş parçacığı çalışmaya hazır.

Adım 3 / 11

ÇIKTI

1) NEW

uyu(ms) ve **katil(t)** : **Thread.sleep** ve **t.join()** çağrılarını try-catch ile saran yardımcı metotlar. Bekleme süreleri sayesinde çıktı hep böyledir.

Durumları Gözlemlemek: Adım Adım

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

main

TIMED_WAITING

uyuyan

TIMED_WAITING

bekleyen

henüz yok

engellenen

henüz yok

İkisi de `sleep` içinde: main 100 ms, uyuyan 1000 ms.

Adım 4 / 11

ÇIKTI

1) NEW

`uyu(ms)` ve `katil(t)` : `Thread.sleep` ve `t.join()` çağrılarını try-catch ile saran yardımcı metotlar. Bekleme süreleri sayesinde çıktı hep böyledir.

Durumları Gözlemlemek: Adım Adım

```
1 public static void main(String[] args) {  
2     Thread uyuyan = new Thread(() -> uyu(1000));  
3     System.out.println("1) " + uyuyan.getState());  
4     uyuyan.start();  
5     uyu(100);  
6     System.out.println("2) " + uyuyan.getState());  
7     Thread bekleyen = new Thread(() -> katil(uyuyan));  
8     bekleyen.start();  
9     uyu(100);  
10    System.out.println("3) " + bekleyen.getState());  
11    Thread engellenen = new Thread(() -> {  
12        synchronized (kilit) { }  
13    });  
14    synchronized (kilit) {  
15        engellenen.start();  
16        uyu(100);  
17        System.out.println("4) " + engellenen.getState());  
18    }  
19    katil(bekleyen);  
20    System.out.println("5) " + uyuyan.getState());  
21 }
```

main

RUNNABLE

uyuyan

TIMED_WAITING

bekleyen

henüz yok

engellenen

henüz yok

main uyandı; uyuyan hâlâ uyuyor:

TIMED_WAITING .

Adım 5 / 11

ÇIKTI

- 1) NEW
- 2) TIMED_WAITING

`uyu(ms)` ve `katil(t)` : `Thread.sleep` ve `t.join()` çağrılarını try-catch ile saran yardımcı metotlar. Bekleme süreleri sayesinde çıktı hep böyledir.

Durumları Gözlemlemek: Adım Adım

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

main

RUNNABLE

uyuyan

TIMED_WAITING

bekleyen

WAITING

engellenen

henüz yok

bekleyen , uyuyan.join() çağırır: süresiz bekler, WAITING .

Adım 6 / 11

ÇIKTI

- 1) NEW
- 2) TIMED_WAITING

`uyu(ms)` ve `katil(t)` : `Thread.sleep` ve `t.join()` çağrılarını try-catch ile saran yardımcı metotlar. Bekleme süreleri sayesinde çıktı hep böyledir.

Durumları Gözlemlemek: Adım Adım

```
1 public static void main(String[] args) {  
2     Thread uyuyan = new Thread(() -> uyu(1000));  
3     System.out.println("1) " + uyuyan.getState());  
4     uyuyan.start();  
5     uyu(100);  
6     System.out.println("2) " + uyuyan.getState());  
7     Thread bekleyen = new Thread(() -> katil(uyuyan));  
8     bekleyen.start();  
9     uyu(100);  
10    System.out.println("3) " + bekleyen.getState());  
11    Thread engellenen = new Thread(() -> {  
12        synchronized (kilit) { }  
13    });  
14    synchronized (kilit) {  
15        engellenen.start();  
16        uyu(100);  
17        System.out.println("4) " + engellenen.getState());  
18    }  
19    katil(bekleyen);  
20    System.out.println("5) " + uyuyan.getState());  
21 }
```

main

RUNNABLE

uyuyan

TIMED_WAITING

bekleyen

WAITING

engellenen

henüz yok

100 ms sonra durum yazdırılır.

Adım 7 / 11

ÇIKTI

- 1) NEW
- 2) TIMED_WAITING
- 3) WAITING

`uyu(ms)` ve `katil(t)` : `Thread.sleep` ve `t.join()` çağrılarını try-catch ile saran yardımcı metotlar. Bekleme süreleri sayesinde çıktı hep böyledir.

Durumları Gözlemlemek: Adım Adım

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

main

RUNNABLE

uyuyan

TIMED_WAITING

bekleyen

WAITING

engellenen

BLOCKED

main **kilit** 'i tutuyor; **engellenen** aynı kilide girmek istiyor: **BLOCKED** .

Adım 8 / 11

ÇIKTI

- 1) NEW
- 2) TIMED_WAITING
- 3) WAITING

uyu(ms) ve **katil(t)** : **Thread.sleep** ve **t.join()** çağrılarını try-catch ile saran yardımcı metotlar. Bekleme süreleri sayesinde çıktı hep böyledir.

Durumları Gözlemlemek: Adım Adım

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

main

RUNNABLE

uyuyan

TIMED_WAITING

bekleyen

WAITING

engellenen

BLOCKED

Kilit hâlâ main'de; durum yazdırılır.

Adım 9 / 11

ÇIKTI

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

`uyu(ms)` ve `katil(t)`: `Thread.sleep` ve `t.join()` çağrılarını try-catch ile saran yardımcı metotlar. Bekleme süreleri sayesinde çıktı hep böyledir.

Durumları Gözlemlemek: Adım Adım

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

main

WAITING

uyuyan

TERMINATED

bekleyen

TERMINATED

engellenen

TERMINATED

Blok bitince kilit bırakılır, engellenen girip biter.
main **bekleyen** 'i bekler; uyuyan 1 sn dolunca biter, bekleyen de biter.

Adım 10 / 11

ÇIKTI

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

uyu(ms) ve **katil(t)** : **Thread.sleep** ve **t.join()** çağrılarını try-catch ile saran yardımcı metotlar. Bekleme süreleri sayesinde çıktı hep böyledir.

Durumları Gözlemlemek: Adım Adım

```
1 public static void main(String[] args) {  
2     Thread uyuyan = new Thread(() -> uyu(1000));  
3     System.out.println("1) " + uyuyan.getState());  
4     uyuyan.start();  
5     uyu(100);  
6     System.out.println("2) " + uyuyan.getState());  
7     Thread bekleyen = new Thread(() -> katil(uyuyan));  
8     bekleyen.start();  
9     uyu(100);  
10    System.out.println("3) " + bekleyen.getState());  
11    Thread engellenen = new Thread(() -> {  
12        synchronized (kilit) { }  
13    });  
14    synchronized (kilit) {  
15        engellenen.start();  
16        uyu(100);  
17        System.out.println("4) " + engellenen.getState());  
18    }  
19    katil(bekleyen);  
20    System.out.println("5) " + uyuyan.getState());  
21 }
```

main

RUNNABLE

uyuyan

TERMINATED

bekleyen

TERMINATED

engellenen

TERMINATED

`run()` 'ı biten iş parçacığı: `TERMINATED` .

Adım 11 / 11

ÇIKTI

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

`uyu(ms)` ve `katil(t)` : `Thread.sleep` ve `t.join()` çağrılarını try-catch ile saran yardımcı metotlar. Bekleme süreleri sayesinde çıktı hep böyledir.

Runnable Arayüzü

- Her iş parçacığı, çalıştırdığı bir `Runnable` nesneye sahiptir.
- `Runnable` arayüzü yalnızca `run()` metodunu içerir.
- Bu arayüzü gerçekleştiren (implements) sınıf, `run()` metodunu tanımlar: iş parçacığının yapacağı iş.
- Tek soyut metotlu (functional) bir arayüz olduğu için Java 8+ ile lambda ifadesi de yazılabilir.

JDK'DAKİ TANIM

```
package java.lang;

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

GERÇEKLEŞTİREN SINIF

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // İş parçacığının yapması gereken görevler
    }
}
```

JAVA 8+: LAMBDA

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

İş Parçacığını Başlatma

```
public class Worker implements Runnable {  
    // İş parçacığının görevleri burada tanımlanır  
    public void run() {  
        String ad = Thread.currentThread().getName();  
        System.out.println("Worker çalışıyor: " + ad);  
    }  
    public static void main(String[] args) {  
        // İş parçacığını başlatma  
        Worker w = new Worker();  
        Thread t = new Thread(w);  
        t.start();  
        System.out.println("main devam ediyor");  
    }  
}
```

- `Thread` yapıcısına parametre olarak `Runnable` nesne verilir.
- `start()` çağrılarak iş parçacığı başlatılır.
- `run()` doğrudan çağrılmaz: aksi takdirde kod, çağıranın iş parçacığında çalışır.

Çıktı (gerçek bir çalıştırma)

```
main devam ediyor  
Worker çalışıyor: Thread-0
```

Satırların sırası çalıştırmadan çalıştırmaya **değişebilir**:

`start()` döner dönmez main devam eder; yeni iş parçacığının ne zaman çalışacağını zamanlayıcı belirler.

start(): İki Yığın, İki Akış

```
1 public class Worker implements Runnable {  
2     // İş parçacığının görevleri burada tanımlanır  
3     public void run() {  
4         String ad = Thread.currentThread().getName();  
5         System.out.println("Worker çalışıyor: " + ad);  
6     }  
7     public static void main(String[] args) {  
8         // İş parçacığını başlatma  
9         ▶ Worker w = new Worker();  
10        Thread t = new Thread(w);  
11        t.start();  
12        System.out.println("main devam ediyor");  
13    }  
14 }
```

main

Thread-0

main()

yığın yok

RUNNABLE

main çalışıyor; **Worker** nesnesi heap'te oluşturulur. Henüz tek iş parçacığı var.

Adım 1 / 8

DEĞİŞKENLER

w = Worker nesnesi

t = -

ÇIKTI

start(): İki Yığın, İki Akış

```

1 public class Worker implements Runnable {
2     // İş parçacığının görevleri burada tanımlanır
3     public void run() {
4         String ad = Thread.currentThread().getName();
5         System.out.println("Worker çalışıyor: " + ad);
6     }
7     public static void main(String[] args) {
8         // İş parçacığını başlatma
9         Worker w = new Worker();
10        Thread t = new Thread(w);
11        t.start();
12        System.out.println("main devam ediyor");
13    }
14 }

```



Adım 2 / 8

DEĞİŞKENLER

w = Worker nesnesi

t = Thread-0 (NEW)

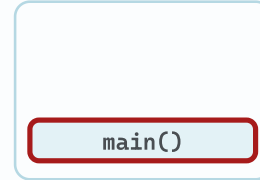
ÇIKTI



start(): İki Yığın, İki Akış

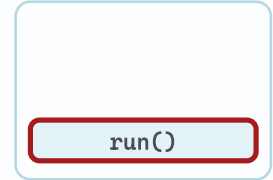
```
1 public class Worker implements Runnable {
2     // İş parçacığının görevleri burada tanımlanır
3     public void run() {
4         String ad = Thread.currentThread().getName();
5         System.out.println("Worker çalışıyor: " + ad);
6     }
7     public static void main(String[] args) {
8         // İş parçacığını başlatma
9         Worker w = new Worker();
10        Thread t = new Thread(w);
11        ▶ t.start();
12        System.out.println("main devam ediyor");
13    }
14 }
```

main



RUNNABLE

Thread-0



RUNNABLE

start() : JVM yeni bir yığın açar ve orada **run()** 'ı başlatır. **start()** hemen döner.

Adım 3 / 8

DEĞİŞKENLER

w = Worker nesnesi

t = Thread-0 (RUNNABLE)

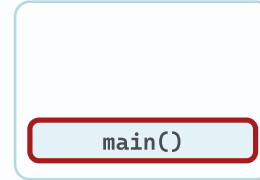
ÇIKTI



start(): İki Yığın, İki Akış

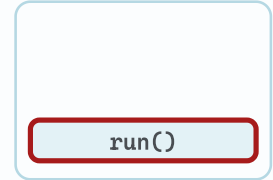
```
1 public class Worker implements Runnable {
2     // İş parçacığının görevleri burada tanımlanır
3     public void run() {
4         String ad = Thread.currentThread().getName();
5         System.out.println("Worker çalışıyor: " + ad);
6     }
7     public static void main(String[] args) {
8         // İş parçacığını başlatma
9         Worker w = new Worker();
10        Thread t = new Thread(w);
11        t.start();
12        System.out.println("main devam ediyor");
13    }
14 }
```

main



RUNNABLE

Thread-0



RUNNABLE (sıra bekliyor)

Bir olası sıra: main önce davranır ve yazdırır.

Adım 4 / 8

DEĞİŞKENLER

w = Worker nesnesi

t = Thread-0 (RUNNABLE)

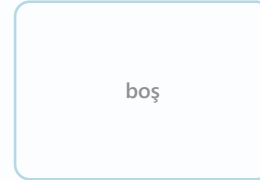
ÇIKTI

main devam ediyor

start(): İki Yığın, İki Akış

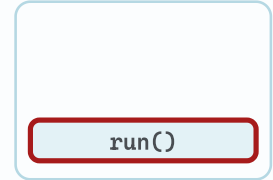
```
1 public class Worker implements Runnable {
2     // İş parçacığının görevleri burada tanımlanır
3     public void run() {
4         String ad = Thread.currentThread().getName();
5         System.out.println("Worker çalışıyor: " + ad);
6     }
7     public static void main(String[] args) {
8         // İş parçacığını başlatma
9         Worker w = new Worker();
10        Thread t = new Thread(w);
11        t.start();
12        System.out.println("main devam ediyor");
13    }
14 }
```

main



TERMINATED

Thread-0



RUNNABLE

`main()` bitti ama program bitmedi: JVM, daemon olmayan iş parçacıkları bitene kadar bekler.

Adım 5 / 8

DEĞİŞKENLER

`t = Thread-0 (RUNNABLE)`

ÇIKTI

main devam ediyor

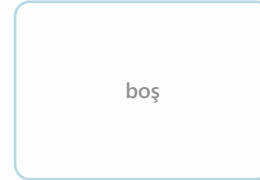
start(): İki Yığın, İki Akış

```

1 public class Worker implements Runnable {
2     // İş parçacığının görevleri burada tanımlanır
3     public void run() {
4         ▶ String ad = Thread.currentThread().getName();
5         System.out.println("Worker çalışıyor: " + ad);
6     }
7     public static void main(String[] args) {
8         // İş parçacığını başlatma
9         Worker w = new Worker();
10        Thread t = new Thread(w);
11        t.start();
12        System.out.println("main devam ediyor");
13    }
14 }

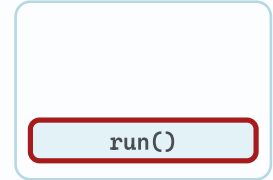
```

main



TERMINATED

Thread-0



RUNNABLE

Thread-0 kendi yığnında çalışır; `ad` onun yerel değişkenidir.

Adım 6 / 8

DEĞİŞKENLER

`t = Thread-0 (RUNNABLE)`

`ad = "Thread-0"`

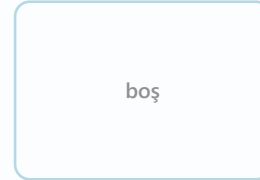
ÇIKTI

main devam ediyor

start(): İki Yığın, İki Akış

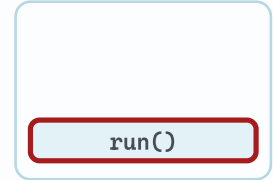
```
1 public class Worker implements Runnable {
2     // İş parçacığının görevleri burada tanımlanır
3     public void run() {
4         String ad = Thread.currentThread().getName();
5         ▶ System.out.println("Worker çalışıyor: " + ad);
6     }
7     public static void main(String[] args) {
8         // İş parçacığını başlatma
9         Worker w = new Worker();
10        Thread t = new Thread(w);
11        t.start();
12        System.out.println("main devam ediyor");
13    }
14 }
```

main



TERMINATED

Thread-0



RUNNABLE

Thread-0 yazdırır.

Adım 7 / 8

DEĞİŞKENLER

t = Thread-0 (RUNNABLE)

ad = "Thread-0"

ÇIKTI

```
main devam ediyor
Worker çalışıyor: Thread-0
```

start(): İki Yığın, İki Akış

```
1 public class Worker implements Runnable {  
2     // İş parçacığının görevleri burada tanımlanır  
3     public void run() {  
4         String ad = Thread.currentThread().getName();  
5         System.out.println("Worker çalışıyor: " + ad);  
6     }  
7     public static void main(String[] args) {  
8         // İş parçacığını başlatma  
9         Worker w = new Worker();  
10        Thread t = new Thread(w);  
11        t.start();  
12        System.out.println("main devam ediyor");  
13    }  
14 }
```

main

Thread-0

boş

boş

TERMINATED

TERMINATED

run() bitti: Thread-0 TERMINATED . Çalışan iş parçacığı kalmadı, JVM kapanır.

Adım 8 / 8

DEĞİŞKENLER

t = Thread-0 (TERMINATED)

ÇIKTI

```
main devam ediyor  
Worker çalışıyor: Thread-0
```

Thread Sınıfını Genişletme

```
public class Worker2 extends Thread {  
    // İş parçacığının görevleri burada tanımlanır  
    public void run() {  
        String ad = getName();    // Thread'den kalıtıldı  
        System.out.println("Worker çalışıyor: " + ad);  
    }  
    public static void main(String[] args) {  
        // İş parçacığını oluşturma ve başlatma  
        Worker2 w = new Worker2();  
        w.start();  
        System.out.println("main devam ediyor");  
    }  
}
```

- İş parçacığı, `Thread` sınıfı genişletilerek (extends) de oluşturulabilir.
- `Runnable` gerçekleştirmek yerine `Thread` alt sınıfı yazılır; `Thread` zaten `Runnable` 'ı gerçekleştirir.
- Nesnenin kendisi bir `Thread` olduğu için doğrudan `w.start()` ve `getName()` çağrılır.

Çıktı (gerçek bir çalışma)

```
main devam ediyor  
Worker çalışıyor: Thread-0
```

Sıra yine değişebilir.

OLUŞTURMA VE BAŞLATMA

start() mı, run() mı?

```
public class StartRunFarki {  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread t = new Thread() -> {  
            String ad = Thread.currentThread().getName();  
            System.out.println("run() içinde: " + ad);  
        };  
        t.run();    // YANLIŞ: main çalıştırır  
        t.start();  // DOĞRU: yeni iş parçasığı  
        t.join();  
    }  
}
```

Çıktı

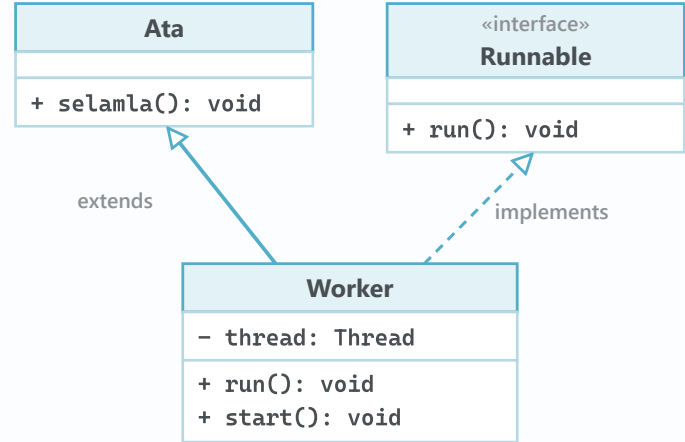
run() içinde: main
run() içinde: Thread-0

Çağrı	Ne olur?
t.run()	Sıradan bir metot çağırısı: aynı iş parçasığında (main) çalışır, yeni yığın yok, bitene kadar main bekler.
t.start()	JVM yeni bir iş parçasığı ve yığın oluşturur; run() orada çalışır, start() hemen döner.
t.start() ikinci kez	IllegalThreadStateException : bir Thread nesnesi yalnızca bir kez başlatılabilir.

Kural: iş parçasığının **işini** run() içine yazarız, iş parçasığını **başlatmak** için start() çağırırız.

Tek Kalıtım Sorunu

- Java bir sınıfın yalnızca **tek bir sınıftan** kalıtım almasına izin verir.
- Çalıştırılabilir sınıf zaten başka bir sınıftan (`Ata`) türemişse, `Thread` sınıfını **genişletemez**: `extends Ata, Thread` derlenmez.
- Çözüm: `Runnable` arayüzünü gerçekleştir ve sınıf içinde `Thread` tipinde bir alan tut.
- Arayüz sayısında sınır yoktur; bu yüzden `Runnable` genellikle tercih edilir.



Tek Kalıtım: Çözüm Kodu

```
class Ata {
    public void selamla() {
        System.out.println("Ata sınıfından merhaba");
    }
}

class Worker extends Ata implements Runnable {
    private Thread thread; // Thread alan olarak
    public void run() {
        String ad = Thread.currentThread().getName();
        System.out.println("run(): " + ad);
    }
    public void start() {
        thread = new Thread(this);
        thread.start();
    }
}

public class TekMirasOrnegi {
    public static void main(String[] args) {
        Worker w = new Worker();
        w.selamla(); // Ata'dan kalıtılan metot
        w.start(); // kendi start() metodumuz
    }
}
```

Çıktı

Ata sınıfından merhaba
run(): Thread-0

- **Worker** hem **Ata** 'nın metotlarını kalıtır hem de bir iş parçacığında çalışabilir.
- Kendi **start()** metodumuz **new Thread(this)** ile kendini bir **Thread** 'e verir.
- **this** , **Runnable** tipindendir; **Thread** yapıcısı bunu kabul eder.

Üç Yol: Karşılaştırma

```
public class UcYol {  
    static class Gorev implements Runnable {        // 1)  
        public void run() {  
            System.out.println("1) Runnable sınıfı");  
        }  
    }  
    static class IsParcacigi extends Thread {        // 2)  
        public void run() {  
            System.out.println("2) Thread alt sınıfı");  
        }  
    }  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread t1 = new Thread(new Gorev());  
        Thread t2 = new IsParcacigi();  
        // 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();  
    }  
}
```

Çıktı

- 1) Runnable sınıfı
- 2) Thread alt sınıfı
- 3) Lambda

Yol	Ne zaman?
<code>implements Runnable</code>	Sınıf başka sınıftan türemişse; iş ile iş parçacığı ayrı tutulur
<code>extends Thread</code>	Kısa, bağımsız örnekler; kalıtım hakkı harcanır
Lambda (Java 8+)	Kısa görevler için en pratik yol

Her `start()` 'tan hemen sonra `join()` geldiği için çıktı sırası sabittir. `static class` : başka sınıfın içinde tanımlanan sınıf.

İki İş Parçacığı: Bir Olası Sıra

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

main



A

B

zaman →

main, A'yı başlatır. A artık **RUNNABLE** ; ne zaman çalışacağı belli değil.

Adım 1 / 8

ÇIKTI



Olası sıralardan biri. Gerçek bir çalıştırma: **B1 · B2 · A1 · A2** . Kesin olan: "main: bitti" en sonudur.

İki İş Parçacığı: Bir Olası Sıra

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

main



A

B

zaman →

main, B'yi başlatır.

Adım 2 / 8

ÇIKTI

Olası sıralardan biri. Gerçek bir çalıştırma: **B1 · B2 · A1 · A2** . Kesin olan: "main: bitti" en sonudur.

İki İş Parçacığı: Bir Olası Sıra

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

main



A

B

zaman →

a.join() : main, A bitene kadar

WAITING : Artık A ve B çalışır.

Adım 3 / 8

ÇIKTI

Olası sıralardan biri. Gerçek bir çalıştırma: B1 · B2 · A1 · A2 . Kesin olan: "main: bitti" en sonudur.

İki İş Parçacığı: Bir Olası Sıra

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

main



A

1

B

zaman →

Zamanlayıcı A'yı çalıştırır: A kendi yığnında
gorev() 'i yürütür.

Adım 4 / 8

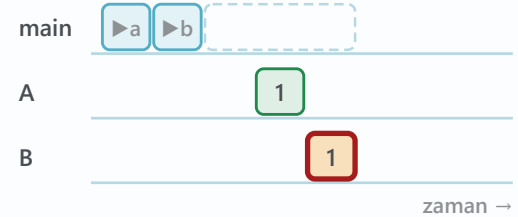
ÇIKTI

A: adım 1

Olası sıralardan biri. Gerçek bir çalıştırma: B1 · B2 · A1 · A2 . Kesin olan: "main: bitti" en sonudur.

İki İş Parçacığı: Bir Olası Sıra

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



Bağlam değişimi (context switch): şimdi B çalışıyor. B'nin `i` 'si ayrı bir yerel değişkendir.

Adım 5 / 8

ÇIKTI

```
A: adım 1  
B: adım 1
```

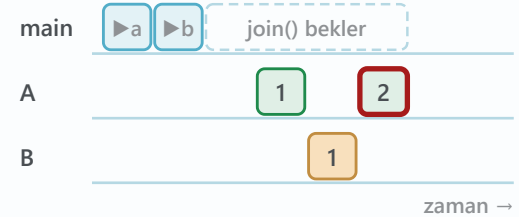
Olası sıralardan biri. Gerçek bir çalıştırma: `B1 · B2 · A1 · A2`. Kesin olan: "main: bitti" en sonudur.

İki İş Parçacığı: Bir Olası Sıra

```

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

```



Tekrar A.

Adım 6 / 8

ÇIKTI

```

A: adım 1
B: adım 1
A: adım 2

```

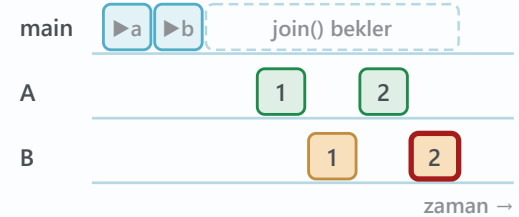
Olası sıralardan biri. Gerçek bir çalıştırma: B1 · B2 · A1 · A2 . Kesin olan: "main: bitti" en sonudur.

İki İş Parçacığı: Bir Olası Sıra

```

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

```



Tekrar B. Ardından ikisi de biter
(**TERMINATED**).

Adım 7 / 8

ÇIKTI

```

A: adım 1
B: adım 1
A: adım 2
B: adım 2

```

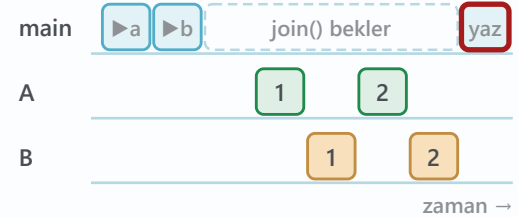
Olası sıralardan biri. Gerçek bir çalıştırma: **B1 · B2 · A1 · A2** . Kesin olan: "main: bitti" en sondadır.

İki İş Parçacığı: Bir Olası Sıra

```

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

```



a.join() döner, b.join() B zaten bittiği için hemen döner; main yazdırır.

Adım 8 / 8

ÇIKTI

```

A: adım 1
B: adım 1
A: adım 2
B: adım 2
main: bitti

```

Olası sıralardan biri. Gerçek bir çalıştırma: B1 · B2 · A1 · A2 . Kesin olan: "main: bitti" en sondadır.

İş Parçacığı Yönetimi: sleep, join, yield

sleep(ms)

Çalışan iş parçacığını belirli bir süre **uyutur** (`TIMED_WAITING`). Kilit tutuyorsa bırakmaz.

join()

Çağırılan iş parçacığı, hedef iş parçacığı **bitene kadar bekler**.
`join(ms)` : en çok ms kadar.

yield()

Yürütmeyi geçici olarak başka iş parçacığına bırakma **ipucu**; zamanlayıcı yok sayabilir.

```
public class JoinOrnegi {  
    static long sonuc = 0;  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread hesap = new Thread(() -> {  
            long toplam = 0;  
            for (int i = 1; i <= 1_000_000; i++) {  
                toplam += i;  
            }  
            sonuc = toplam;  
        });  
        hesap.start();  
        System.out.println("join öncesi : " + sonuc);  
        hesap.join(); // hesap bitene kadar bekle  
        System.out.println("join sonrası : " + sonuc);  
    }  
}
```

Çıktı

join öncesi : 0
join sonrası: 500000500000

`join()` olmadan main, hesap bitmeden `sonuc` 'u okur: **0** görür. `join()` sonrası hesap kesinlikle bitmiştir ve yazdığı değer görünür.

`sleep` ve `join`,

`InterruptedException`

fırlatabilir: yakalanmalı ya da

`throws` ile bildirilmelidir.

Arka Plan (Daemon) İş Parçacıkları

```
public class DaemonOrnegi {  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread saat = new Thread(() -> {  
            int s = 0;  
            while (true) {  
                System.out.println("tik " + s++);  
                try {  
                    Thread.sleep(300);  
                } catch (InterruptedException e) {  
                    return;  
                }  
            }  
        });  
        saat.setDaemon(true);    // start()'tan ÖNCE  
        saat.start();  
        Thread.sleep(1000);  
        System.out.println("main bitti, JVM kapanıyor");  
    }  
}
```

Çıktı

```
tik 0  
tik 1  
tik 2  
tik 3  
main bitti, JVM kapanıyor
```

- JVM, **daemon olmayan** tüm iş parçacıkları bitince kapanır; daemon iş parçacıklarının bitmesi beklenmez.
- `saat` sonsuz döngüde olsa da main bitince program sonlanır.
- `setDaemon(true)` , `start()` 'tan **önce** çağrılmalı; sonra çağrılırsa `IllegalThreadStateException` .

İş Parçacığını Durdurma

- İş parçacığı, `run()` metodu tamamlandığında durur.
- JVM kapanır: `System.exit()` (içeride `Runtime.exit()`) çağrıldığında ya da daemon olmayan tüm iş parçacıkları bittiğinde.
- Bir iş parçacığını dışarıdan durdurmanın doğru yolu **iş birliğidir**: iş parçacığı periyodik olarak bir **bayrak** (flag) kontrol eder.
- Bayrak, değişikliği diğer iş parçacıklarının görebilmesi için `volatile` tanımlanır.
- Alternatif: `interrupt()` ile kesme isteği (ileride).

Thread.stop() kullanmayın

Kullanımdan kaldırılmış (deprecated) `stop()`, iş parçacığını işinin ortasında öldürür; tuttuğu kilitler bırakılır ve nesneler yarım güncellenmiş kalabilir. Java 20+'da artık çalışmaz:

```
Thread t = new Thread(() -> { });
try {
    t.stop();
} catch (UnsupportedOperationException e) {
    System.out.println("stop(): desteklenmiyor");
}
```

Çıktı (JDK 21)

```
stop(): desteklenmiyor
```

Bayrak ile Durdurma

```
class Worker extends Thread {  
    private volatile boolean flag = false;  
    public void finish() { flag = true; }  
    public void run() {  
        while (!flag) {  
            System.out.println("çalışıyor...");  
            try { Thread.sleep(100); }  
            catch (InterruptedException e) { return; }  
        }  
        System.out.println("bayrak görüldü, bitiyor");  
    }  
}  
  
public class BayrakDurdurma {  
    public static void main(String[] args)  
        throws InterruptedException {  
        Worker w = new Worker();  
        w.start();  
        Thread.sleep(350);  
        w.finish();    // bayrağı kaldır  
        w.join();       // bitmesini bekle  
        System.out.println("main: işçi durdu");  
    }  
}
```

Çıktı

```
çalışıyor...  
çalışıyor...  
çalışıyor...  
çalışıyor...  
bayrak görüldü, bitiyor  
main: işçi durdu
```

- **finish()** yalnızca bayrağı kaldırır; iş parçacığı döngünün başında bunu görür ve **run()** 'dan **kendisi** çıkar.
- main 350 ms uyurken işçi 0, 100, 200, 300 ms'de dört kez yazdı.
- **volatile** olmasaydı işçi bayrağın değiştiğini hiç görmeyebilirdi.

Referans ile Durdurma

```
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 ben = Thread.currentThread();  
        while (thread == ben) {    // ... iş ...  
        }  
        System.out.println("run() bitti");  
    }  
}  
  
public class ReferansDurdurma {  
    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: bitti");  
    }  
}
```

Çıktı

```
run() bitti  
main: bitti
```

- Sınıf `Runnable` gerçekleştirir; çalışan `Thread` nesnesi `volatile` bir alanda tutulur.
- Döngü, alan hâlâ **kendi** iş parçasığını gösterdikçe sürer: `Thread.currentThread()` ile karşılaştırılır.
- `finish()` alanı `null` yapar; koşul bozulur, `run()` biter.
- `currentThread()` `static` olduğundan `Thread` 'i genişletmeyen sınıfta `Thread.` ön eki şarttır.

İşlemleri Doğru Sırayla Gerçekleştirmek

Hız belirsiz

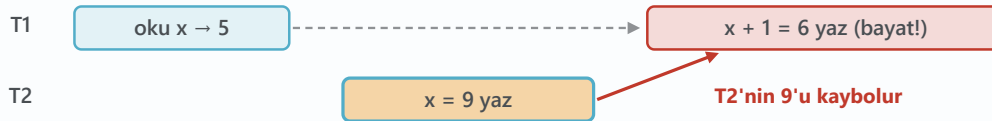
Her iş parçacığının çalışma hızı ve sıklığı değişkendir. İşletim sistemleri öncelikleri **görmezden gelebilir**; sıraya öncelikle güvenilmez.

Bayat değer

Bir iş parçacığı değişkeni okuduktan sonra başkası değeri güncelleyebilir; ilki **güncel olmayan** değerle çalışmaya devam eder.

Yeniden sıralama

Derleyici, JIT ve işlemci, optimizasyon için komutların sırasını değiştirebilir; tek iş parçacığında fark edilmez, birden fazlasında beklenmeyen davranış doğurur.



Paylaşılan ve **değişen** veriye birden fazla iş parçacığı erişiyorsa erişim **senkronize** edilmelidir. Aksi hâlde sonuç, iş parçacıklarının hangi sırayla çalıştığına bağlı olur: buna **yarış durumu** (race condition) denir.

Yarış Durumu (Race Condition)

```
public class YarisDurumu {  
    static int sayac = 0;    // paylaşılan veri  
    public static void main(String[] args)  
        throws InterruptedException {  
        Runnable arttir = () -> {  
            for (int i = 0; i < 100_000; i++) {  
                sayac++;    // oku, 1 ekle, yaz: atomik DEĞİL  
            }  
        };  
        Thread t1 = new Thread(arttir);  
        Thread t2 = new Thread(arttir);  
        t1.start();  
        t2.start();  
        t1.join();  
        t2.join();  
        System.out.println("Beklenen: 200000");  
        System.out.println("Bulunan : " + sayac);  
    }  
}
```

1. çalıştırma

Beklenen: 200000

Bulunan : 138001

2. çalıştırma

Beklenen: 200000

Bulunan : 135352

3. çalıştırma

Beklenen: 200000

Bulunan : 136776

Her çalıştırmada **farklı** ve 200000'den küçük bir sonuç: bazı artışlar kayboluyor.

sayac++ Neden Kaybolur?

```
► // sayac++ işlemcide üç adımdır:  
2 int yazmac = sayac;    // 1) oku  
3 yazmac = yazmac + 1;   // 2) 1 ekle  
4 sayac = yazmac;        // 3) yaz
```

bellek
sayac = 0

t1
yazmac = -

t2
yazmac = -

Başta `sayac = 0` . İki iş parçacığı da `sayac++` yapacak; doğru sonuç 2 olmalı.

Adım 1 / 7

DEĞİŞKENLER

sayac = 0

t1.yazmac = -

t2.yazmac = -

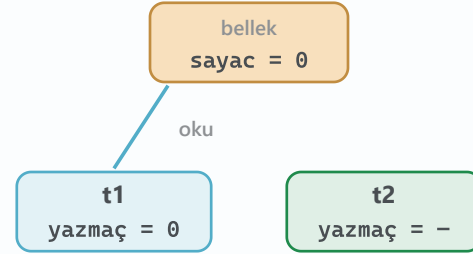
ÇAĞRI YIĞINI

boş

Bu, olası sıralardan yalnızca biridir; araya girme olmazsa sonuç doğru çıkar. Hata bu yüzden **rastgele** görünür ve test etmesi zordur. Çağrı yığını kutusu o anda çalışan iş parçacığını gösterir.

sayac++ Neden Kaybolur?

```
1 // sayac++ işlemcide üç adımdır:  
▶ int yazmac = sayac; // 1) oku  
3 yazmac = yazmac + 1; // 2) 1 ekle  
4 sayac = yazmac; // 3) yaz
```



t1 okur: yazmacına 0 gelir.

Adım 2 / 7

DEĞİŞKENLER

sayac = 0

t1.yazmac = 0

t2.yazmac = -

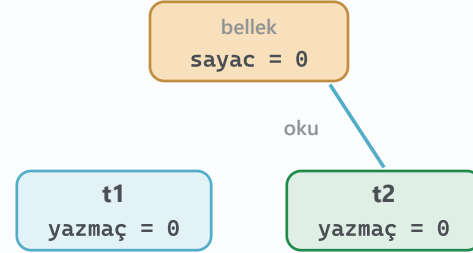
ÇAĞRI YIĞINI

t1

Bu, olası sıralardan yalnızca biridir; araya girme olmazsa sonuç doğru çıkar. Hata bu yüzden **rastgele** görünür ve test etmesi zordur. Çağrı yığını kutusu o anda çalışan iş parçacığını gösterir.

sayac++ Neden Kaybolur?

```
1 // sayac++ işlemcide üç adımdır:  
▶ int yazmac = sayac; // 1) oku  
3 yazmac = yazmac + 1; // 2) 1 ekle  
4 sayac = yazmac; // 3) yaz
```



Bağlam değişimi! t2 de okur: o da 0 görür.

Adım 3 / 7

DEĞİŞKENLER

sayac = 0

t1.yazmac = 0

t2.yazmac = 0

ÇAĞRI YIĞINI

t2

Bu, olası sıralardan yalnızca biridir; araya girme olmazsa sonuç doğru çıkar. Hata bu yüzden **rastgele** görünür ve test etmesi zordur. Çağrı yığını kutusu o anda çalışan iş parçacığını gösterir.

sayac++ Neden Kaybolur?

```
1 // sayac++ işlemcide üç adımdır:  
2 int yazmac = sayac;    // 1) oku  
3 ► yazmac = yazmac + 1; // 2) 1 ekle  
4 sayac = yazmac;        // 3) yaz
```

bellek
sayac = 0

t1
yazmac = 1

t2
yazmac = 0

t1 kendi yazmacında 1 ekler.

Adım 4 / 7

DEĞİŞKENLER

sayac = 0

t1.yazmac = 1

t2.yazmac = 0

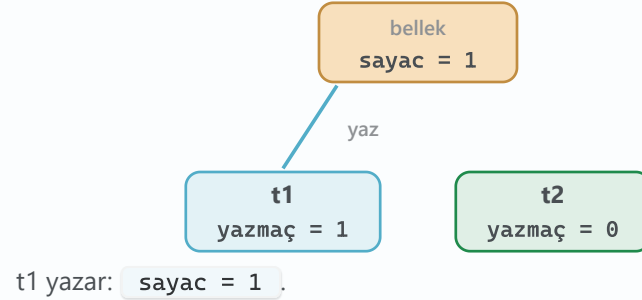
ÇAĞRI YIĞINI

t1

Bu, olası sıralardan yalnızca biridir; araya girme olmazsa sonuç doğru çıkar. Hata bu yüzden **rastgele** görünür ve test etmesi zordur. Çağrı yığını kutusu o anda çalışan iş parçacığını gösterir.

sayac++ Neden Kaybolur?

```
1 // sayac++ işlemcide üç adımdır:  
2 int yazmac = sayac;    // 1) oku  
3 yazmac = yazmac + 1;   // 2) 1 ekle  
▶ sayac = yazmac;        // 3) yaz
```



Adım 5 / 7

DEĞİŞKENLER

sayac = 1

t1.yazmac = 1

t2.yazmac = 0

ÇAĞRI YIĞINI

t1

Bu, olası sıralardan yalnızca biridir; araya girme olmazsa sonuç doğru çıkar. Hata bu yüzden **rastgele** görünür ve test etmesi zordur. Çağrı yığını kutusu o anda çalışan iş parçacığını gösterir.

sayac++ Neden Kaybolur?

```
1 // sayac++ işlemcide üç adımdır:  
2 int yazmac = sayac;      // 1) oku  
3 ► yazmac = yazmac + 1;   // 2) 1 ekle  
4 sayac = yazmac;          // 3) yaz
```

bellek
sayac = 1

t1
yazmac = 1

t2
yazmac = 1

t2 kendi (eski) değerine 1 ekler: 1.

Adım 6 / 7

DEĞİŞKENLER

sayac = 1

t1.yazmac = 1

t2.yazmac = 1

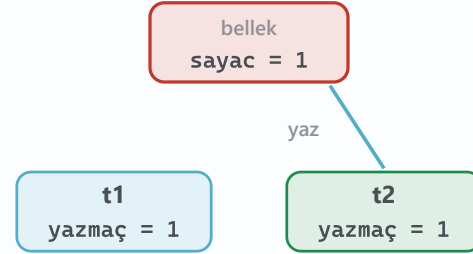
ÇAĞRI YIĞINI

t2

Bu, olası sıralardan yalnızca biridir; araya girme olmazsa sonuç doğru çıkar. Hata bu yüzden **rastgele** görünür ve test etmesi zordur. Çağrı yığını kutusu o anda çalışan iş parçacığını gösterir.

sayac++ Neden Kaybolur?

```
1 // sayac++ işlemcide üç adımdır:  
2 int yazmac = sayac;    // 1) oku  
3 yazmac = yazmac + 1;   // 2) 1 ekle  
▶ sayac = yazmac;        // 3) yaz
```



t2 yazar: `sayac = 1` . İki artış yapıldı ama sonuç 1: t1'in artışı **kayboldu**.

Adım 7 / 7

DEĞİŞKENLER

sayac = 1

t1.yazmac = 1

t2.yazmac = 1

ÇAĞRI YIĞINI

t2

Bu, olası sıralardan yalnızca biridir; araya girme olmazsa sonuç doğru çıkar. Hata bu yüzden **rastgele** görünür ve test etmesi zordur. Çağrı yığını kutusu o anda çalışan iş parçacığını gösterir.

Senkronizasyon Seçenekleri

join()

Bir iş parçacığının bitmesini beklemek.

synchronized

Nesnenin monitör kilidiyle karşılıklı dışlama.

Lock nesneleri

`ReentrantLock` ,
`ReadWriteLock` : açık kilitleme.

Bariyer, semafor, değiştirici

`CyclicBarrier` ,
`Semaphore` , `Exchanger` ,
`CountDownLatch` .

volatile değişkenler

Okuma/yazmanın tüm iş parçacıklarınca görünmesi.

Atomik nesneler

`AtomicInteger` gibi bölünemez işlemler.

Senkronizasyona benzeyen ama olmayanlar: öncelikler (priority), uyutma (`sleep`), yol verme (`yield`) ve zamanlayıcılar (timer). Bunlar sırayı **garanti etmez**; "biraz bekleyeyim, öbürü bitirmiş olur" diye kod yazılmaz.

synchronized ve Monitör

- Her nesnenin, kilitlenip açılabilen bir **monitörü** (monitor, iç kilit) vardır.
- Monitöre aynı anda yalnızca **bir** iş parçacığı sahip olabilir.
- t1 monitöre sahipken t2 de isterse t2 bekler (**BLOCKED**).
- Monitör bırakılınca bekleyenler **rekabet eder**, yalnızca biri sahip olur.
- Monitör **synchronized** ile alınır; blok bitince (istisna ile çıkılsa bile) otomatik bırakılır.
- Böylece **karşılıklı dışlama** (mutual exclusion) sağlanır, veri bütünlüğü korunur.

SÖZ DİZİMİ

```
synchronized (o) {  
    // o'nun monitörü bu blokta bizde  
}
```

```
Runnable r = () -> {  
    for (int i = 0; i < 50_000; i++) {  
        synchronized (o) {    // o'nun monitörünü al  
            ortak++;  
        }                      // monitör bırakılır  
    }  
};
```

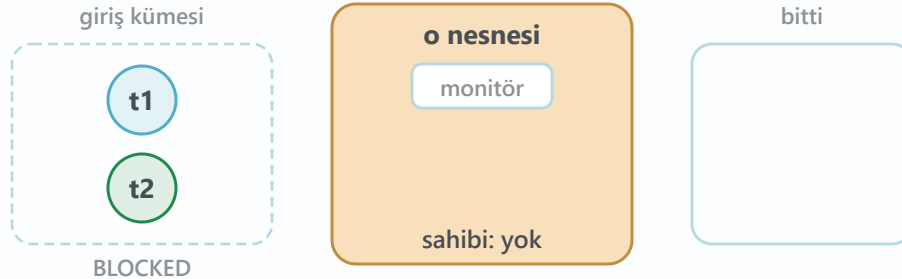
Çıktı (2 × 50000, her seferinde)

100000

Monitör Adım Adım

```
// t1 ve t2 aynı kodu çalıştırır  
synchronized (o) {  
    ortak++;    // kritik bölge  
}
```

Kritik bölge (critical section): paylaşılan veriye erişen ve aynı anda yalnızca bir iş parçacığının çalıştırması gereken kod parçası.



t1 ve t2 aynı anda **synchronized (o)** satırına gelir; monitör boştur.

Monitör Adım Adım

```
// t1 ve t2 aynı kodu çalıştırır  
synchronized (o) {  
    ortak++;    // kritik bölge  
}
```

Kritik bölge (critical section): paylaşılan veriye erişen ve aynı anda yalnızca bir iş parçacığının çalıştırması gereken kod parçası.

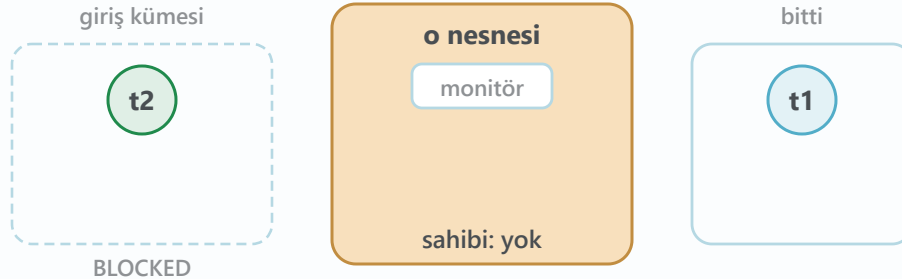


t1 kazanır ve monitörü alır. t2 giriş kümesinde **BLOCKED** bekler.

Monitör Adım Adım

```
// t1 ve t2 aynı kodu çalıştırır  
synchronized (o) {  
    ortak++;    // kritik bölge  
}
```

Kritik bölge (critical section): paylaşılan veriye erişen ve aynı anda yalnızca bir iş parçacığının çalıştırması gereken kod parçası.



t1 bloktan çıkar: monitör otomatik bırakılır. Bekleyenler rekabet eder.

Monitör Adım Adım

```
// t1 ve t2 aynı kodu çalıştırır  
synchronized (o) {  
    ortak++;    // kritik bölge  
}
```

Kritik bölge (critical section): paylaşılan veriye erişen ve aynı anda yalnızca bir iş parçacığının çalıştırması gereken kod parçası.

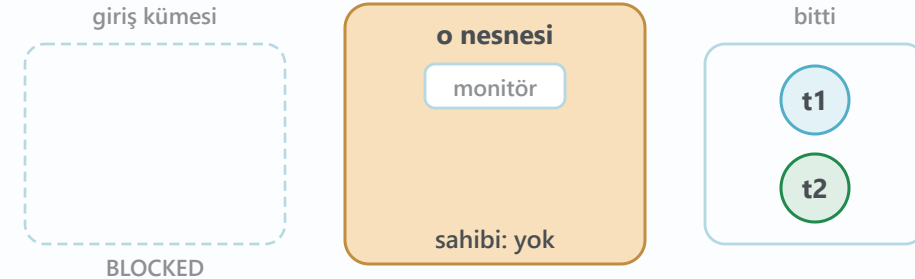


t2 monitörü alır ve kritik bölgeye girer.

Monitör Adım Adım

```
// t1 ve t2 aynı kodu çalıştırır  
synchronized (o) {  
    ortak++;    // kritik bölge  
}
```

Kritik bölge (critical section): paylaşılan veriye erişen ve aynı anda yalnızca bir iş parçacığının çalıştırması gereken kod parçası.



t2 de çıkar. **ortak++** iki kez, birbirine karışmadan yapıldı.

Metotları Senkronize Tanımlama

`synchronized` METOT

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

EŞDEĞERİ: `synchronized` BLOK

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

- Metodu `synchronized` tanımlamak, gövdesini bir `synchronized` bloğa almakla **aynıdır**.
- Örnek metot → `this` nesnesinin monitörü.
- `static` metot → sınıfın `C.class` nesnesinin monitörü (nesne yok).
- Bu yüzden bir `static synchronized` ile bir örnek `synchronized` metot birbirini **engellemez**: farklı kilitler.
- Blok biçimi, yalnızca gerçekten gereken satırları kitlemeye izin verir.

`...` yerine gövde yazıldığında (örneğin `n++`, `s++`) iki biçim derlenir ve aynı davranır.

Yarış Önmek

```
class Sayac {  
    private int deger = 0;  
    public synchronized void arttir() { deger++; }  
    public synchronized int getDeger() { return deger; }  
}  
  
public class SenkronSayac {  
    public static void main(String[] args)  
        throws InterruptedException {  
        Sayac sayac = new Sayac();  
        Runnable gorev = () -> {  
            for (int i = 0; i < 100_000; i++) {  
                sayac.arttir();  
            }  
        };  
        Thread t1 = new Thread(gorev);  
        Thread t2 = new Thread(gorev);  
        t1.start();  
        t2.start();  
        t1.join();  
        t2.join();  
        System.out.println("Sonuç: " + sayac.getDeger());  
    }  
}
```

Çıktı (her çalıştırmada)

Sonuç: 200000

- `arttir()` artık `synchronized` : oku-ekle-yaz adımları bölünmeden yapılır.
- Okuyan `getDeger()` da senkronize olmalı; yoksa güncel değeri görmesi garanti değildir.
- Bedeli: iş parçacıkları sırayla girer, biraz yavaşlar. Yalnızca paylaşılan veriyi koruyun.

Karşılıklı Dışlama: OncelikliKuyruk

```
class OncelikliKuyruk {
    private int[] veri;
    private int boyut;
    public OncelikliKuyruk(int n) { veri = new int[n]; }
    public synchronized void ekle(int o) {
        if (boyut == veri.length)
            throw new HeapFullException();
        veri[boyut++] = o;
    }
    public synchronized int cikar() { // en küçüğü
        int enk = 0;
        for (int i = 1; i < boyut; i++) {
            if (veri[i] < veri[enk]) enk = i;
        }
        int sonuc = veri[enk];
        veri[enk] = veri[--boyut];
        return sonuc;
    }
    public synchronized String toString() {
        return Arrays.toString(Arrays.copyOf(veri, boyut));
    }
}
```

```
OncelikliKuyruk k =
    new OncelikliKuyruk(3);
k.ekle(5);
k.ekle(2);
k.ekle(8);
System.out.println(k);
int x = k.cikar();
System.out.println(x);
System.out.println(k);
k.ekle(1);
try {
    k.ekle(9);
} catch (HeapFullException e) {
    String m = e.getMessage();
    System.out.println(m);
}
```

Çıktı

```
[5, 2, 8]
2
[5, 8]
Kuyruk dolu
```

Tüm metotlar aynı monitörü (`this`) kullanır.

volatile Alanlar

- Performans için değerler önbellekte ya da **yazmaçta** (register) tutulabilir; o zaman diğer iş parçacıkları güncel değere erişemez.
- `volatile`, değişkenin her okunuşta **ana bellekten** okunmasını, her yazılışta hemen görünür olmasını garanti eder.
- `long` ve `double` dışındaki değişkenlerin tek okuma/yazması zaten atomiktir; `volatile long/double` da atomik olur.
- Paylaşılan basit bir bayrak için `synchronized` ve kilit gerektirmez.
- Ama yalnızca okuma ve yazma atomiktir: `x++` gibi oku-değiştir-yaz işlemi **atomik değildir**.



Görünürlük Sorunu

```
public class GorunurlukSorunu {  
    static boolean dur = false; // volatile DEĞİL!  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread t = new Thread(() -> {  
            long tur = 0;  
            while (!dur) {  
                tur++;  
            }  
            System.out.println("iş parçacığı durdu");  
        });  
        t.setDaemon(true);  
        t.start();  
        Thread.sleep(500);  
        dur = true;  
        System.out.println("main: dur = true");  
        t.join(2000); // en çok 2 sn bekle  
        boolean canli = t.isAlive();  
        System.out.println("hâlâ çalışıyor mu? " + canli);  
    }  
}
```

Çıktı: volatile YOK

main: dur = true
hâlâ çalışıyor mu? true

Çıktı: static volatile boolean dur

main: dur = true
iş parçacığı durdu
hâlâ çalışıyor mu? false

`volatile` olmadan JIT derleyicisi `dur` 'u bir kez okuyup döngüyü `while (true)` gibi çalıştırabilir: iş parçacığı değişikliği **hiç görmez**. Sonuç JVM'e göre değişebilir; doğruluk şansa bırakılmaz.

`setDaemon(true)` : takılan iş parçacığı programın kapanmasını engellemesin diye.

volatile Atomik Değildir, final ile Birleşmez

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

Üç çalıştırma (beklenen 200000)

```
volatile sayac: 163299
volatile sayac: 164441
volatile sayac: 199332
```

Görünürlük tamam, ama iki iş parçacığı yine aynı değeri okuyup aynı sonucu yazabiliyor. Çözüm:

`synchronized` ya da `AtomicInteger` .

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

Derleyici hatası

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

final	volatile
Bir kez atanır (yapıcıda); sonra değişmez	Değeri değişebilir
Yapıcı bitince herkes aynı değeri görür	Her okuma güncel değeri döndürür

İkisi senkronizasyon ihtiyacını farklı yoldan giderir; birlikte kullanılamaz.

Java Memory Model: happens-before

Java Bellek Modeli (JMM), bir iş parçacığının yazdığı değeri ötekinin **ne zaman** göreceğini tanımlar. Programcı işlemci bariyerleriyle değil, **önce-olur** (happens-before) ilişkileriyle düşünür: A önce-olur B ise, A'nın yazdıkları B'de kesinlikle görünür.

Kural	A önce-olur B
Kilit	bir monitörün/kilidin bırakılması → aynı kilidin sonraki alınması
<code>volatile</code>	bir <code>volatile</code> alana yazma → aynı alanın sonraki okunması
Başlatma	<code>t.start()</code> çağrısından önceki her şey → <code>t</code> 'nin içindeki işlemler
Bitirme	<code>t</code> 'nin bütün işlemleri → <code>t.join()</code> 'den dönen iş parçacığı
Hazır yapılar	<code>BlockingQueue.put</code> → aynı öğeyi alan <code>take</code> ; <code>Future</code> 'ın işi → <code>get()</code>

Yanlış paylaşım (false sharing): farklı iş parçacıklarının **bağımsız** değişkenleri aynı ön bellek satırında (çoğunlukla 64 bayt) yer alıp sık güncellenirse çekirdekler satırı birbirinden sürekli çeker. Sonuç yanlış olmaz ama program **yavaşlar**; bu bir doğruluk değil, başarım sorunudur.

Açık Kilitleme (Explicit Locking)

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

- Karmaşık senkronizasyon senaryoları için kullanılır.
- `java.util.concurrent.locks` paketindeki `Lock` arayüzü; `ReentrantLock` ve `ReadWriteLock` gelişmiş, özelleştirilebilir kilitler sunar.
- Esneklik ve performans iyileştirmeleri sağlar: denemeli, süreli, kesilebilir kilit alma.
- Kullanırken dikkatli olunmalı: kilidi **siz** açmalısınız.
- İçer aktarma: `java.util.concurrent.locks.*`

Çıktı

Sonuç: 200000

AÇIK KİLİTLER

Kilidi finally Bloğunda Aç

DOĞRU KALIP

```
public void calistir() {  
    Lock kilit = ...;  
    kilit.lock();  
    try {  
        // korunan kaynağa erişim  
    } finally {  
        kilit.unlock();  
    }  
}
```

- `lock()` , `try` 'dan hemen **önce** çağrılır.
- `unlock()` `finally` içindedir: kritik bölgede istisna olsa bile kilit **her durumda** bırakılır.

YANLIŞ: `finally` YOK

```
kilit.lock();  
islemYap();    // istisna fırlatırsa...  
kilit.unlock(); // ...bu satıra hiç gelinmez
```

Kilit kilitli kalır; onu isteyen her iş parçacığı sonsuza dek bekler. `synchronized` 'da bu sorun yoktur: blok bitince kilit kendiliğinden bırakılır.

YANLIŞ: ALINMAMIŞ KİLİDİ AÇMAK

```
Lock kilit = new ReentrantLock();  
kilit.unlock(); // hiç alınmamış kilit
```

Çalışma zamanı hatası

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

tryLock(): Beklemeden ya da Süreli Kilit Alma

```
static void calistir()
    throws InterruptedException {
    if (kilit.tryLock()) {
        // alındı
        try { System.out.println("1: alındı"); }
        finally { kilit.unlock(); }
    } else {
        // alınamadı, başka bir şey yap
        System.out.println("1: alınamadı");
    }
    if (kilit.tryLock(3, TimeUnit.SECONDS)) {
        try { System.out.println("2: alındı"); }
        finally { kilit.unlock(); }
    } else {
        // 3 saniye içinde alınamadı
        System.out.println("2: alınamadı");
    }
}
```

- `tryLock()` : kilit boşsa alır, `true` ; doluyrsa **beklemeden** `false` .
- `tryLock(3, SECONDS)` : en çok 3 sn bekler; `InterruptedException` bildirilmelidir.

```
kilit.lock(); // kilidi main tutuyor
Thread t = new Thread(() -> {
    try { calistir(); }
    catch (InterruptedException e) { }
});
t.start();
Thread.sleep(1000);
System.out.println("main: kilidi bırakıyor");
kilit.unlock();
t.join();
```

Çıktı

```
1: alınamadı
main: kilidi bırakıyor
2: alındı
```

ReentrantLock Sınıfı

- Adı “yeniden girişe izin veren” (reentrant) anlamına gelir.
- Kilidi tutan iş parçacığı **aynı kilidi tekrar** alabilir; beklemez, yalnızca bir sayaç artar. Kilit, sayaç 0'a inince bırakılır.
- Birden fazla `ReentrantLock` kullanılarak iç içe kritik bölgelere erişilebilir.
- **Adil kilitleme** (fair lock): `new ReentrantLock(true)` ile kilit, en uzun bekleyene verilir.
- `synchronized` da yeniden girişlidir: `synchronized` bir metot aynı nesnenin başka bir `synchronized` metodunu çağırabilir.

```
ReentrantLock kilit1 = new ReentrantLock();
// true: adil (fair) kilit
ReentrantLock kilit2 = new ReentrantLock(true);
// ...
kilit1.lock();
try {
    // kritik bölgeye erişim
    kilit2.lock();
    try {
        // iç içe kritik bölgeye erişim
        System.out.println("iki kilit de alındı");
    } finally { kilit2.unlock(); }
} finally { kilit1.unlock(); }
System.out.println("adil mi? " + kilit2.isFair());
```

Çıktı

```
iki kilit de alındı
adil mi? true
```

Yeniden Giriş: Adım Adım

```
1 import java.util.concurrent.locks.ReentrantLock;
2
3 public class YenidenGiris {
4     static final ReentrantLock kilit =
5         new ReentrantLock();
6     static void yaz(String yer) {
7         System.out.println(yer + kilit.getHoldCount());
8     }
9     static void dis() {
10         kilit.lock();
11         try { yaz("dis(): "); ic(); }
12         finally { kilit.unlock(); }
13     }
14     static void ic() {
15         kilit.lock(); // beklemeden girer
16         try { yaz("ic(): "); }
17         finally { kilit.unlock(); }
18     }
19     public static void main(String[] args) {
20         ▶ dis();
21         yaz("main: ");
22     }
23 }
```

kilit

sahibi: yok



sayı = 0

main, **dis()** 'i çağırır. Kilit boş.

Adım 1 / 8

ÇAĞRI YIĞINI

main()

ÇIKTI



Yeniden Giriş: Adım Adım

```
1 import java.util.concurrent.locks.ReentrantLock;
2
3 public class YenidenGiris {
4     static final ReentrantLock kilit =
5         new ReentrantLock();
6     static void yaz(String yer) {
7         System.out.println(yer + kilit.getHoldCount());
8     }
9     static void dis() {
10         ▶ kilit.lock();
11         try { yaz("dis(): "); ic(); }
12         finally { kilit.unlock(); }
13     }
14     static void ic() {
15         kilit.lock(); // beklemeden girer
16         try { yaz("ic(): "); }
17         finally { kilit.unlock(); }
18     }
19     public static void main(String[] args) {
20         dis();
21         yaz("main: ");
22     }
23 }
```

kilit

sahibi: main



sayı = 1

dis() kilidi alır: sahibi main, tutma sayısı 1.

Adım 2 / 8

ÇAĞRI YIĞINI

dis()

main()

ÇIKTI



Yeniden Giriş: Adım Adım

```
1 import java.util.concurrent.locks.ReentrantLock;
2
3 public class YenidenGiris {
4     static final ReentrantLock kilit =
5         new ReentrantLock();
6     static void yaz(String yer) {
7         System.out.println(yer + kilit.getHoldCount());
8     }
9     static void dis() {
10        kilit.lock();
11        ▶ try { yaz("dis(): "); ic(); }
12            finally { kilit.unlock(); }
13    }
14    static void ic() {
15        kilit.lock(); // beklemeden girer
16        try { yaz("ic(): "); }
17            finally { kilit.unlock(); }
18    }
19    public static void main(String[] args) {
20        dis();
21        yaz("main: ");
22    }
23 }
```

kilit

sahibi: main



sayı = 1

yaz sayıyı yazdırır.

Adım 3 / 8

ÇAĞRI YIĞINI

yaz()

dis()

main()

ÇIKTI

dis(): 1

Yeniden Giriş: Adım Adım

```
1 import java.util.concurrent.locks.ReentrantLock;
2
3 public class YenidenGiris {
4     static final ReentrantLock kilit =
5         new ReentrantLock();
6     static void yaz(String yer) {
7         System.out.println(yer + kilit.getHoldCount());
8     }
9     static void dis() {
10        kilit.lock();
11        try { yaz("dis(): "); ic(); }
12        finally { kilit.unlock(); }
13    }
14    static void ic() {
15        ▶ kilit.lock(); // beklemeden girer
16        try { yaz("ic(): "); }
17        finally { kilit.unlock(); }
18    }
19    public static void main(String[] args) {
20        dis();
21        yaz("main: ");
22    }
23 }
```

kilit

sahibi: main



sayı = 2

ic() aynı kilidi ister. Sahibi zaten main olduğu için **beklemez**: sayı 2 olur.

Adım 4 / 8

ÇAĞRI YIĞINI

ic()

dis()

main()

ÇIKTI

dis(): 1

Yeniden Giriş: Adım Adım

```
1 import java.util.concurrent.locks.ReentrantLock;
2
3 public class YenidenGiris {
4     static final ReentrantLock kilit =
5         new ReentrantLock();
6     static void yaz(String yer) {
7         System.out.println(yer + kilit.getHoldCount());
8     }
9     static void dis() {
10         kilit.lock();
11         try { yaz("dis(): "); ic(); }
12         finally { kilit.unlock(); }
13     }
14     static void ic() {
15         kilit.lock(); // beklemeden girer
16         ▶ try { yaz("ic(): "); }
17         finally { kilit.unlock(); }
18     }
19     public static void main(String[] args) {
20         dis();
21         yaz("main: ");
22     }
23 }
```

kilit

sahibi: main



sayı = 2

Yazdırılır.

Adım 5 / 8

ÇAĞRI YIĞINI

yaz()

ic()

dis()

main()

ÇIKTI

dis(): 1

ic(): 2

Yeniden Giriş: Adım Adım

```
1 import java.util.concurrent.locks.ReentrantLock;
2
3 public class YenidenGiris {
4     static final ReentrantLock kilit =
5         new ReentrantLock();
6     static void yaz(String yer) {
7         System.out.println(yer + kilit.getHoldCount());
8     }
9     static void dis() {
10        kilit.lock();
11        try { yaz("dis(): "); ic(); }
12        finally { kilit.unlock(); }
13    }
14    static void ic() {
15        kilit.lock(); // beklemeden girer
16        try { yaz("ic(): "); }
17        finally { kilit.unlock(); }
18    }
19    public static void main(String[] args) {
20        dis();
21        yaz("main:  ");
22    }
23 }
```

kilit

sahibi: main



sayı = 1

ic() 'nin finally 'si: unlock() sayıyı
1'e indirir; kilit hâlâ main'de.

Adım 6 / 8

ÇAĞRI YIĞINI

ic()

dis()

main()

ÇIKTI

dis(): 1

ic(): 2

Yeniden Giriş: Adım Adım

```
1 import java.util.concurrent.locks.ReentrantLock;
2
3 public class YenidenGiris {
4     static final ReentrantLock kilit =
5         new ReentrantLock();
6     static void yaz(String yer) {
7         System.out.println(yer + kilit.getHoldCount());
8     }
9     static void dis() {
10         kilit.lock();
11         try { yaz("dis(): "); ic(); }
12         finally { kilit.unlock(); }
13     }
14     static void ic() {
15         kilit.lock(); // beklemeden girer
16         try { yaz("ic(): "); }
17         finally { kilit.unlock(); }
18     }
19     public static void main(String[] args) {
20         dis();
21         yaz("main: ");
22     }
23 }
```

kilit

sahibi: yok



sayı = 0

dis() 'in finally 'si: sayı 0, kilit gerçekten bırakıldı.

Adım 7 / 8

ÇAĞRI YIĞINI

dis()

main()

ÇIKTI

dis(): 1

ic(): 2

Yeniden Giriş: Adım Adım

```
1 import java.util.concurrent.locks.ReentrantLock;
2
3 public class YenidenGiris {
4     static final ReentrantLock kilit =
5         new ReentrantLock();
6     static void yaz(String yer) {
7         System.out.println(yer + kilit.getHoldCount());
8     }
9     static void dis() {
10         kilit.lock();
11         try { yaz("dis(): "); ic(); }
12         finally { kilit.unlock(); }
13     }
14     static void ic() {
15         kilit.lock(); // beklemeden girer
16         try { yaz("ic(): "); }
17         finally { kilit.unlock(); }
18     }
19     public static void main(String[] args) {
20         dis();
21         yaz("main: ");
22     }
23 }
```

kilit

sahibi: yok



sayı = 0

Her **lock()** için bir **unlock()** gerekir.

Adım 8 / 8

ÇAĞRI YIĞINI**yaz()**

main()

ÇIKTI

```
dis(): 1
ic(): 2
main: 0
```

ReadWriteLock Sınıfı



Okuma kilidi

Birden fazla iş parçacığı veriyi **aynı anda** okuyabilir; okurken birbirini engellemez (paralel okuma).

Yazma kilidi

Bir iş parçacığı yazarken diğerlerinin okuması da yazması da engellenir; yazma güvenli ve atomik olur.

Ne zaman?

Verinin **sık okunup nadiren yazıldığı** durumlarda (önbellek, ayar tablosu) performansı artırır.

ReadWriteLock Kodu

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

public String oku(String anahtar) {
    kilit.readLock().lock(); // okuma kilidi al
    try {
        return tablo.get(anahtar); // veriyi oku
    } finally {
        kilit.readLock().unlock();
    }
}

public void yaz(String anahtar, String deger) {
    kilit.writeLock().lock(); // yazma kilidi al
    try {
        tablo.put(anahtar, deger); // veriyi yaz
    } finally {
        kilit.writeLock().unlock();
    }
}
```

```
AyarTablosu a = new AyarTablosu();
a.yaz("dil", "tr");
Runnable okuyucu = () -> System.out.println(
    Thread.currentThread().getName()
    + ": " + a.oku("dil"));
Thread o1 = new Thread(okuyucu, "Okuyucu-1");
Thread o2 = new Thread(okuyucu, "Okuyucu-2");
o1.start(); o2.start();
o1.join(); o2.join();
```

Çıktı (bir çalıştırma)

```
Okuyucu-2: tr
Okuyucu-1: tr
```

İki okuyucu okuma kilidini aynı anda tutabilir; sıraları değişebilir.

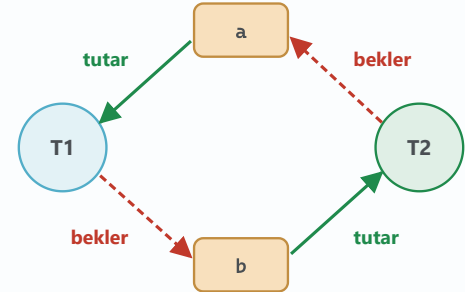
ReentrantReadWriteLock ,
ReadWriteLock arayüzünün
gerçekleştirmesidir.

Açık (Explicit) ve Dolaylı (Implicit) Kilitleme

	Açık: Lock	Dolaylı: synchronized
Kontrol	Geliştirici <code>lock()</code> / <code>unlock()</code> çağırır	Otomatik: blok girişinde alınır, çıkışta bırakılır
Sınıflar	<code>Lock</code> arayüzü, <code>ReentrantLock</code> , <code>ReentrantReadWriteLock</code>	Anahtar kelime; her nesnenin monitörü
Beklemeden deneme	<code>tryLock()</code> , <code>tryLock(süre)</code> : engellemeden koşullu alma	Yok: kilit boşalana kadar bekler
Kesilebilir bekleme	<code>lockInterruptibly()</code>	Yok
Adillik, koşullar	<code>new ReentrantLock(true)</code> , birden çok <code>Condition</code>	Yok; tek bekleme kümesi
Yeniden giriş	<code>ReentrantLock</code> yeniden girişlidir	Her zaman yeniden girişlidir
Risk	<code>unlock()</code> unutulursa kilit sonsuza dek kalır	Unutma riski yok
Ne zaman?	Gelişmiş kontrol gerektiğinde	Basit senaryolar, daha okunur kod

Kilitlenme (Deadlock)

```
static void calis(Object x, Object y, String ad) {  
    synchronized (x) {          // 1. kilit  
        uyu(100);  
        synchronized (y) {      // 2. kilit  
            System.out.println(ad + " bitti");  
        }  
    }  
}  
  
public static void main(String[] args)  
    throws InterruptedException {  
    Thread t1 = new Thread(() -> calis(a, b, "T1"));  
    Thread t2 = new Thread(() -> calis(b, a, "T2"));  
    t1.setDaemon(true);  
    t2.setDaemon(true);  
    t1.start();  
    t2.start();  
    Thread.sleep(500);  
    String d = t1.getState() + " " + t2.getState();  
    System.out.println(d);  
    long[] k = ManagementFactory.getThreadMXBean().  
        .findDeadlockedThreads();  
    int n = (k == null) ? 0 : k.length;  
    System.out.println("Kilitlenmiş: " + n);  
}
```



Çıktı

BLOCKED BLOCKED
Kilitlenmiş: 2

İki ya da daha fazla iş parçasığının, birbirinin tuttuğu kilidi **sonsuz** **dek** beklemesi. Kilitler **farklı sırada** alındığında oluşur. `uyu(ms)` :

`sleep` 'i try-catch ile sarar.

Kilitlenme Adım Adım

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



Adım 1 / 8

DEĞİŞKENLER

a = Object

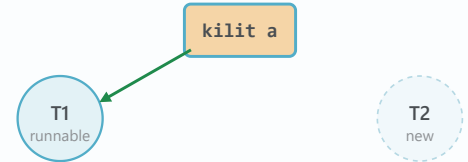
b = Object

ÇAĞRI YIĞINI

main()

Kilitlenme Adım Adım

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



yeşil: kilidi tutuyor · kırmızı: bekliyor

T1 başlar ve `synchronized (a)` ile a'yı alır.

Adım 2 / 8

DEĞİŞKENLER

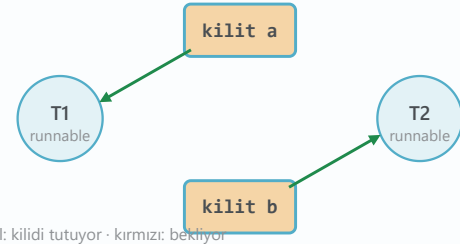
T1: `x = a, y=b`

ÇAĞRI YIĞINI

T1: `calis(a, b)`

Kilitlenme Adım Adım

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



T2 ters sırayla çağrılmıştır: önce **b**'yi alır.
Şimdilik sorun yok.

Adım 3 / 8

DEĞİŞKENLER

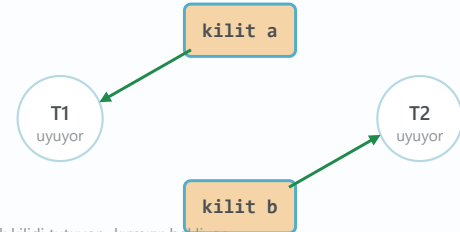
T2: $x = b, y = a$

ÇAĞRI YİĞİNİ

T2: `calis(b, a)`

Kilitlenme Adım Adım

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



yeşil: kilidi tutuyor · kırmızı: bekliyor

İkisi de 100 ms uyur; `sleep` tuttuğu kilidi bırakmaz.

Adım 4 / 8

DEĞİŞKENLER

a = T1 tutuyor

b = T2 tutuyor

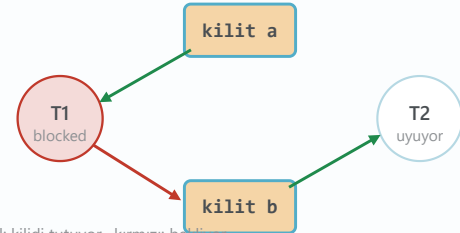
ÇAĞRI YIĞINI

T2: uyu(100)

T1: uyu(100)

Kilitlenme Adım Adım

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



T1 uyanır, iç blok için **b**'yi ister. b, T2'de: T1
BLOCKED .

Adım 5 / 8

DEĞİŞKENLER

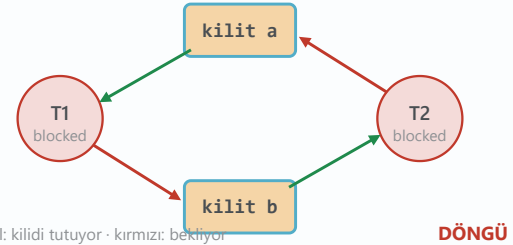
T1 istiyor = b

ÇAĞRI YİĞİNİ

T1: calis(a, b)

Kilitlenme Adım Adım

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



T2 de **a**'yı ister; a, T1'de. Her biri ötekini bıraktığını bekler: **döngü** oluştu.

DÖNGÜ

Adım 6 / 8

DEĞİŞKENLER

T2 istiyor = a

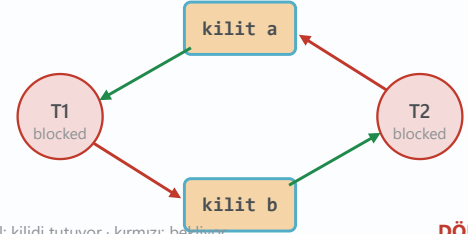
ÇAĞRI YİĞİNİ

T2: calis(b, a)

T1: calis(a, b)

Kilitlenme Adım Adım

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



yeşil: kilidi tutuyor · kırmızı: bekliyor

DÖNGÜ

"bitti" satırına hiçbiri ulaşamaz; program sonsuza dek bekler.

`findDeadlockedThreads()` 2 iş parçasığı bildirir.

Adım 7 / 8

DEĞİŞKENLER

T1 = BLOCKED

T2 = BLOCKED

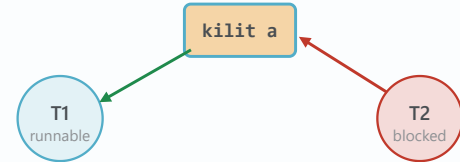
ÇAĞRI YİĞİNİ

T2: calis(b, a)

T1: calis(a, b)

Kilitlenme Adım Adım

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



yeşil: kilidi tutuyor · kırmızı: bekliyor

İki iş parçasığı da kilitleri **aynı sırayla** (önce a) alsaydı T2 a'da beklerdi, b'yi hiç tutmazdı: döngü kurulamaz.

Adım 8 / 8

DEĞİŞKENLER

T2 çağrısı = `calis(a, b, "T2")`

ÇAĞRI YİĞİNİ

T2: `calis(a, b)`

T1: `calis(a, b)`

Kilitlenmeyi Önlemek

TEK DEĞİŞİKLİK: AYNI SIRA

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

Çıktı

```
T1 bitti  
T2 bitti  
TERMINATED TERMINATED  
Kilitlenmiş: 0
```

Herkes önce **a** 'yı, sonra **b** 'yi alır: döngü oluşamaz. T2, **a** 'yı bekler; T1 bitince alır ve o da biter.

Sabit kilit sırası

Tüm iş parçacıkları kilitleri **aynı sırayla** alsın (ör. kimlik numarasına göre).

Sürelili deneme

`tryLock(süre)` alamazsa elindekileri bırakıp sonra tekrar dener.

Kısa kritik bölge

Kilit tutarken başka kilit istemekten ve uzun işlerden kaçın.

Teşhis

`jstack` ya da `ThreadMXBean.findDeadlockedThreads()` kilitlenmeyi gösterir.

Canlı Kilit (Livelock) ve Açlık (Starvation)

Kilitlenme

herkes blokludur, hiçbir şey değişmez

Canlı kilit

herkes çalışıyor, ama ilerleme yok

Açlık

biri sürekli sıra bekliyor

Canlı kilit

İş parçacıkları blokludur değildir; birbirine sürekli "yol vererek" durum değiştirirler ama hiçbiri işini bitiremez. Örnek: iki iş parçacığı `tryLock` başarısız olunca elindekini bırakıp **aynı anda** yeniden dener ve hep çıkarılır.

```
Lock adil = new ReentrantLock(true); // adil
if (kilit.tryLock(50, TimeUnit.MILLISECONDS)) {
    // ... iş, sonra kilit.unlock()
} else {
    Thread.sleep(rnd.nextInt(20)); // rastgele
}
```

Açlık

Bir iş parçacığı kaynağa uzun süre ya da hiç erişemez: kilidi hep başkaları kapar, ya da uzun kritik bölgeler onu bekletir. Öncelik (`setPriority`) değerleri bunu çözmek için **güvenilir değildir**.

- **Adil kilit** en uzun bekleyene sıra verir (biraz daha yavaştır).
- Yeniden denemeden önce **rastgele** süre beklemek simetriyi bozar, canlı kilidi önler.
- Kritik bölgeleri kısa tutmak herkese sıra gelmesini sağlar.

Durum (Condition) Senkronizasyonu

- Ortak bir kaynağa **yalnızca belirli bir koşul** sağlandığında güvenli erişmeyi sağlar.
- Örnek: tüketici, kuyruk **boş değilken** öge almalı; üretici, kuyruk **dolu değilken** eklemeli.
- Kontrollü güncelleme için nesne kilidi gerekir; ayrıca koşul sağlanana kadar beklemek gerekir.
- Ama **kilidi tutarken beklenemez**: koşulu değiştirecek iş parçasığı kilidi alamaz, herkes takılır.

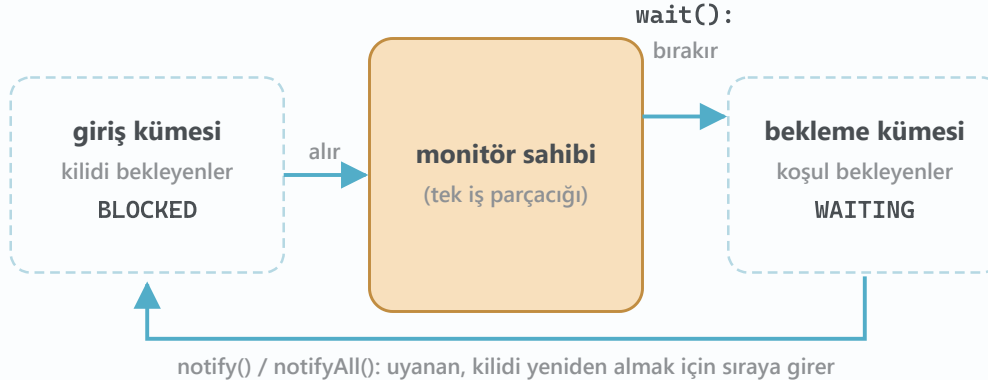
Java'nın destekleri

- 1) `Object.wait()` ve `Object.notify()` / `notifyAll()`
- 2) `Condition` arayüzü (`Lock` ile)
- 3) Senkronizasyon nesneleri (`CountDownLatch` , `Semaphore` ...) ve hazır `BlockingQueue`

YANLIŞ: MEŞGUL BEKLEME

```
while (kuyruk.bosMu()) {  
    // boşuna işlemci yakar; kilit tutuluyorsa  
    // üretici hiç ekleyemez  
}
```

Object.wait() ve Object.notify()



- Her nesnenin bir **bekleme kümesi** (wait set) vardır.
- Kilidi tutan iş parçacığı `wait()` çağırır: kümeye girer, **kilidi bırakır**, bloke olur.
- `notify()` birini, `notifyAll()` hepsini uyandırır; `wait(ms)` süre dolunca da döner.
- Uyanan, kilidi yeniden alınca **kaldığı yerden** devam eder.
- Kural 1: yalnızca `synchronized` içinde çağrılır; yoksa `IllegalMonitorStateException: current thread is not owner`.
- Kural 2: koşul `if` ile değil `while` ile denetlenir: uyanınca koşul yine bozulmuş olabilir (ya da sahte uyanma).

BlockingQueue: wait() ve notifyAll() ile

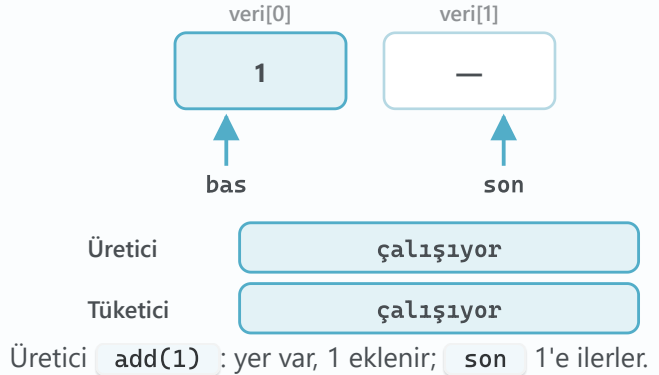
```
class BlockingQueue {
    private Object[] veri;
    private int bas = 0;
    private int son = 0;
    private int buyukluk = 0;
    public BlockingQueue(int boyut) {
        veri = new Object[boyut];
    }
    public synchronized int getSize() {
        return buyukluk;
    }
    public synchronized void add(Object oge)
        throws InterruptedException {
        while (buyukluk == veri.length) {
            wait();
        }
        veri[son] = oge;
        son = (son + 1) % veri.length;
        buyukluk++;
        notifyAll();
    }
}
```

```
public synchronized Object remove()
    throws InterruptedException {
    while (buyukluk == 0) {
        wait();
    }
    Object oge = veri[bas];
    bas = (bas + 1) % veri.length;
    buyukluk--;
    notifyAll();
    return oge;
}
```

- Dairesel (circular) dizi: **bas** çıkarılacak, **son** eklenecek konum; **% veri.length** başa sarar.
- Doluysa **add**, boşsa **remove** bekler; her değişiklikten sonra **notifyAll()** bekleyenleri uyandırır.

Üretici–Tüketici: Adım Adım

```
1 public synchronized void add(Object oge)
2     throws InterruptedException {
3     while (buyukluk == veri.length) {
4         wait();
5     }
6     veri[son] = oge;
7     son = (son + 1) % veri.length;
8     buyukluk++;
9     notifyAll();
10 }
11 public synchronized Object remove()
12     throws InterruptedException {
13     while (buyukluk == 0) {
14         wait();
15     }
16     Object oge = veri[bas];
17     bas = (bas + 1) % veri.length;
18     buyukluk--;
19     notifyAll();
20     return oge;
21 }
```



Adım 1 / 9

DEĞİŞKENLER

bas = 0

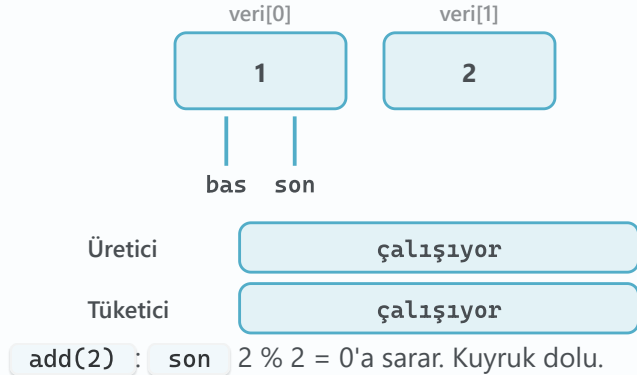
son = 1

buyukluk = 1

Kapasite 2. Bu, olası sıralardan biridir; gerçek sıra zamanlayıcıya bağlıdır ama kuyruk hiçbir sırada taşmaz ya da boşken okunmaz.

Üretici–Tüketici: Adım Adım

```
1 public synchronized void add(Object oge)
2     throws InterruptedException {
3     while (buyukluk == veri.length) {
4         wait();
5     }
6     veri[son] = oge;
7     son = (son + 1) % veri.length;
8     buyukluk++;
9     notifyAll();
10 }
11 public synchronized Object remove()
12     throws InterruptedException {
13     while (buyukluk == 0) {
14         wait();
15     }
16     Object oge = veri[bas];
17     bas = (bas + 1) % veri.length;
18     buyukluk--;
19     notifyAll();
20     return oge;
21 }
```



Adım 2 / 9

DEĞİŞKENLER

bas = 0

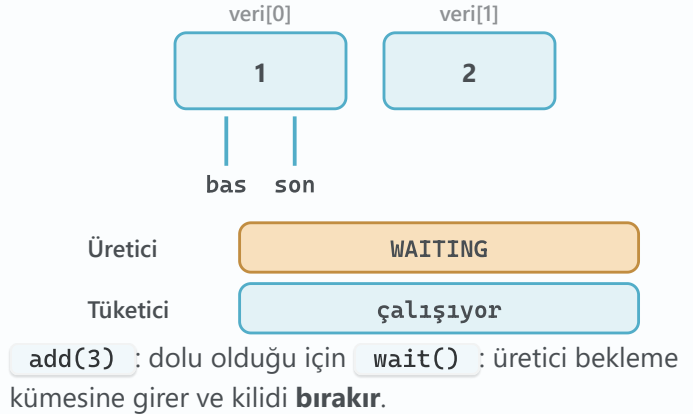
son = 0

buyukluk = 2

Kapasite 2. Bu, olası sıralardan biridir; gerçek sıra zamanlayıcıya bağlıdır ama kuyruk hiçbir sırada taşmaz ya da boşken okunmaz.

Üretici–Tüketici: Adım Adım

```
1 public synchronized void add(Object oge)
2     throws InterruptedException {
3     while (buyukluk == veri.length) {
4         wait();
5     }
6     veri[son] = oge;
7     son = (son + 1) % veri.length;
8     buyukluk++;
9     notifyAll();
10 }
11 public synchronized Object remove()
12     throws InterruptedException {
13     while (buyukluk == 0) {
14         wait();
15     }
16     Object oge = veri[bas];
17     bas = (bas + 1) % veri.length;
18     buyukluk--;
19     notifyAll();
20     return oge;
21 }
```



Adım 3 / 9

DEĞİŞKENLER

bas = 0

son = 0

buyukluk = 2

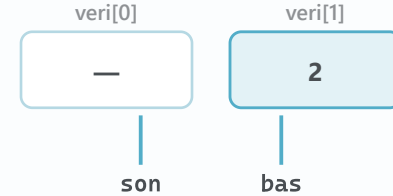
Kapasite 2. Bu, olası sıralardan biridir; gerçek sıra zamanlayıcıya bağlıdır ama kuyruk hiçbir sırada taşmaz ya da boşken okunmaz.

Üretici–Tüketici: Adım Adım

```

1 public synchronized void add(Object oge)
2     throws InterruptedException {
3     while (buyukluk == veri.length) {
4         wait();
5     }
6     veri[son] = oge;
7     son = (son + 1) % veri.length;
8     buyukluk++;
9     notifyAll();
10 }
11 public synchronized Object remove()
12     throws InterruptedException {
13     while (buyukluk == 0) {
14         wait();
15     }
16     ▶ Object oge = veri[bas];
17     ▶ bas = (bas + 1) % veri.length;
18     ▶ buyukluk--;
19     notifyAll();
20     return oge;
21 }

```



Üretici

WAITING

Tüketici

çalışıyor

Tüketici kilidi alabildi: `remove()` 1'i alır, `bas` 1 olur.

Adım 4 / 9

DEĞİŞKENLER

bas = 1

son = 0

buyukluk = 1

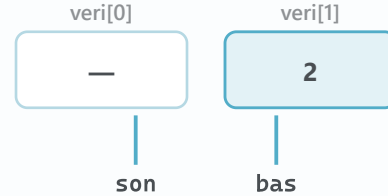
Kapasite 2. Bu, olası sıralardan biridir; gerçek sıra zamanlayıcıya bağlıdır ama kuyruk hiçbir sırada taşmaz ya da boşken okunmaz.

Üretici–Tüketici: Adım Adım

```

1 public synchronized void add(Object oge)
2     throws InterruptedException {
3     while (buyukluk == veri.length) {
4         wait();
5     }
6     veri[son] = oge;
7     son = (son + 1) % veri.length;
8     buyukluk++;
9     notifyAll();
10 }
11 public synchronized Object remove()
12     throws InterruptedException {
13     while (buyukluk == 0) {
14         wait();
15     }
16     Object oge = veri[bas];
17     bas = (bas + 1) % veri.length;
18     buyukluk--;
19     ▶ notifyAll();
20     return oge;
21 }

```



Üretici

BLOCKED

Tüketici

çalışıyor

`notifyAll()` : üretici uyanır ama kilit hâlâ tüketicide; kilidi bekler. `remove` 1 döndürür.

Adım 5 / 9

DEĞİŞKENLER

bas = 1

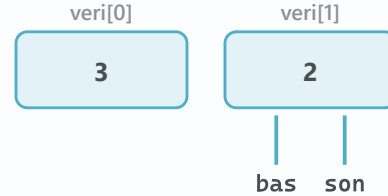
son = 0

buyukluk = 1

Kapasite 2. Bu, olası sıralardan biridir; gerçek sıra zamanlayıcıya bağlıdır ama kuyruk hiçbir sırada taşmaz ya da boşken okunmaz.

Üretici–Tüketici: Adım Adım

```
1 public synchronized void add(Object oge)
2     throws InterruptedException {
3     while (buyukluk == veri.length) {
4         wait();
5     }
6     veri[son] = oge;
7     son = (son + 1) % veri.length;
8     buyukluk++;
9     notifyAll();
10 }
11 public synchronized Object remove()
12     throws InterruptedException {
13     while (buyukluk == 0) {
14         wait();
15     }
16     Object oge = veri[bas];
17     bas = (bas + 1) % veri.length;
18     buyukluk--;
19     notifyAll();
20     return oge;
21 }
```



Üretici

çalışıyor

Tüketici

çalışıyor

Üretici kilidi alır, `while` koşulunu **yeniden** denetler: dolu değil. 3, `veri[0]` 'a yazılır.

Adım 6 / 9

DEĞİŞKENLER

bas = 1

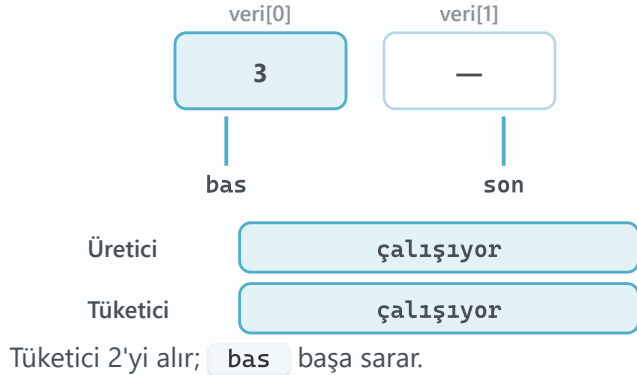
son = 1

buyukluk = 2

Kapasite 2. Bu, olası sıralardan biridir; gerçek sıra zamanlayıcıya bağlıdır ama kuyruk hiçbir sırada taşmaz ya da boşken okunmaz.

Üretici–Tüketici: Adım Adım

```
1 public synchronized void add(Object oge)
2     throws InterruptedException {
3     while (buyukluk == veri.length) {
4         wait();
5     }
6     veri[son] = oge;
7     son = (son + 1) % veri.length;
8     buyukluk++;
9     notifyAll();
10 }
11 public synchronized Object remove()
12     throws InterruptedException {
13     while (buyukluk == 0) {
14         wait();
15     }
16     ▶ Object oge = veri[bas];
17     ▶ bas = (bas + 1) % veri.length;
18     ▶ buyukluk--;
19     notifyAll();
20     return oge;
21 }
```



Adım 7 / 9

DEĞİŞKENLER

bas = 0

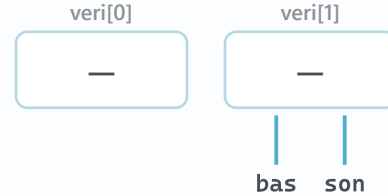
son = 1

buyukluk = 1

Kapasite 2. Bu, olası sıralardan biridir; gerçek sıra zamanlayıcıya bağlıdır ama kuyruk hiçbir sırada taşmaz ya da boşken okunmaz.

Üretici–Tüketici: Adım Adım

```
1 public synchronized void add(Object oge)
2     throws InterruptedException {
3     while (buyukluk == veri.length) {
4         wait();
5     }
6     veri[son] = oge;
7     son = (son + 1) % veri.length;
8     buyukluk++;
9     notifyAll();
10 }
11 public synchronized Object remove()
12     throws InterruptedException {
13     while (buyukluk == 0) {
14         wait();
15     }
16     ▶ Object oge = veri[bas];
17     ▶ bas = (bas + 1) % veri.length;
18     ▶ buyukluk--;
19     notifyAll();
20     return oge;
21 }
```



Üretici

çalışıyor

Tüketici

çalışıyor

Tüketici 3'ü alır. Kuyruk boş.

Adım 8 / 9

DEĞİŞKENLER

bas = 1

son = 1

buyukluk = 0

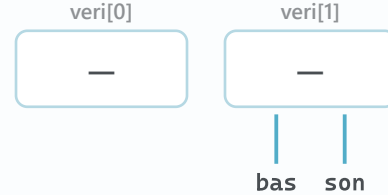
Kapasite 2. Bu, olası sıralardan biridir; gerçek sıra zamanlayıcıya bağlıdır ama kuyruk hiçbir sırada taşmaz ya da boşken okunmaz.

Üretici–Tüketici: Adım Adım

```

1 public synchronized void add(Object oge)
2     throws InterruptedException {
3     while (buyukluk == veri.length) {
4         wait();
5     }
6     veri[son] = oge;
7     son = (son + 1) % veri.length;
8     buyukluk++;
9     notifyAll();
10 }
11 public synchronized Object remove()
12     throws InterruptedException {
13     while (buyukluk == 0) {
14         wait();
15     }
16     Object oge = veri[bas];
17     bas = (bas + 1) % veri.length;
18     buyukluk--;
19     notifyAll();
20     return oge;
21 }

```



Üretici

çalışıyor

Tüketici

WAITING

Bir sonraki `remove()` : boş olduğu için tüketici bekler; üretici 4'ü ekleyip `notifyAll()` yapınca uyanacak.

Adım 9 / 9

DEĞİŞKENLER

bas = 1

son = 1

buyukluk = 0

Kapasite 2. Bu, olası sıralardan biridir; gerçek sıra zamanlayıcıya bağlıdır ama kuyruk hiçbir sırada taşmaz ya da boşken okunmaz.

Üretici–Tüketici: Gerçek Çalıştırma

```
BlockingQueue kuyruk = new BlockingQueue(2);
Thread uretici = new Thread(() -> {
    try {
        for (int i = 1; i <= 4; i++) {
            kuyruk.add(i);
            System.out.println("üretildi: " + i);
        }
    } catch (InterruptedException e) { }
});
Thread tuketici = new Thread(() -> {
    try {
        for (int i = 1; i <= 4; i++) {
            Thread.sleep(100);
            Object x = kuyruk.remove();
            System.out.println("tüketildi: " + x);
        }
    } catch (InterruptedException e) { }
});
uretici.start();
tuketici.start();
uretici.join();
tuketici.join();
System.out.println("kalan: " + kuyruk.getSize());
```

Çıktı (bir çalıştırma)

```
üretildi: 1
üretildi: 2
üretildi: 3
tüketildi: 1
üretildi: 4
tüketildi: 2
tüketildi: 3
tüketildi: 4
kalan: 0
```

Satırların sırası değişebilir.

Burada "üretildi: 3", "tüketildi: 1"den önce görüldü: tüketici

remove 'dan döndükten sonra, yazdıramadan üretici çalıştı.

Condition Arayüzü

- Bir iş parçacığının **durum değişene kadar beklemesini**, durum değişince kaldığı yerden devam etmesini sağlar.
- `wait()` / `notify()` işlevselliğini `Lock` dünyasına taşır; `ReentrantLock` ile birlikte kullanılır:
`kilit.newCondition()`.
- `await()` : kilidi bırakır ve iş parçacığını askıya alır.
- `signal()` / `signalAll()` : bekleyen bir / tüm iş parçacıklarını uyandırır.
- Bir kilide **birden çok koşul** bağlanabilir: üreticiler ve tüketiciler ayrı kuyrukta bekler.

```
class BlockingQueue {  
    private final Lock kilit = new ReentrantLock();  
    private final Condition doluDegil =  
        kilit.newCondition();  
    private final Condition bosDegil =  
        kilit.newCondition();  
    private Object[] veri;  
    private int bas = 0;  
    private int son = 0;  
    private int buyukluk = 0;  
    public BlockingQueue(int boyut) {  
        veri = new Object[boyut];  
    }  
    // add() ve remove() metotları  
}
```

BlockingQueue: Condition ile

```
public void add(Object oge)
    throws InterruptedException {
    kilit.lock();
    try {
        while (buyukluk == veri.length)
            doluDegil.await();
        veri[son] = oge;
        son = (son + 1) % veri.length;
        buyukluk++;
        bosDegil.signal();
    } finally {
        kilit.unlock();
    }
}
```

```
public Object remove()
    throws InterruptedException {
    kilit.lock();
    try {
        while (buyukluk == 0)
            bosDegil.await();
        Object oge = veri[bas];
        bas = (bas + 1) % veri.length;
        buyukluk--;
        doluDegil.signal();
        return oge;
    } finally {
        kilit.unlock();
    }
}
```

Çıktı: üretici 1–4 ekler, main 4 kez çıkarır

1 2 3 4

Ekleyen yalnızca **bosDegil** koşulunu, çıkaran yalnızca **doluDegil** koşulunu uyarır: gereksiz uyandırma yok, bu yüzden **signal()** yeterli.

Hazır Çözüm: java.util.concurrent

```
BlockingQueue<Integer> kuyruk =  
    new ArrayBlockingQueue<>(2);  
Thread uretici = new Thread(() -> {  
    try {  
        // put(): kuyruk doluysa bekler  
        for (int i = 1; i <= 4; i++) kuyruk.put(i);  
    } catch (InterruptedException e) { }  
});  
uretici.start();  
for (int i = 1; i <= 4; i++) {  
    // take(): kuyruk boşsa bekler  
    System.out.print(kuyruk.take() + " ");  
}  
System.out.println();  
uretici.join();
```

Çıktı

1 2 3 4

Yol	Bekleme	Uyandırma
synchronized	wait()	notify() , notifyAll()
Lock + Condition	await()	signal() , signalAll()
ArrayBlockingQueue	put() , take() içinde	otomatik

Gerçek projelerde hazır

`java.util.concurrent.BlockingQueue`

gerçekleştirmeleri

(`ArrayBlockingQueue` ,

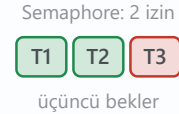
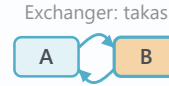
`LinkedBlockingQueue`) tercih edilir;

kendi sınıfımız mekanizmayı anlamak

içindir.

Senkronizasyon Nesneleri

Çoklu iş parçacıkları arasında **düzeni** sağlayan hazır araçlar (`java.util.concurrent`); her biri farklı bir senaryoya çözüm sunar.



CountDownLatch (geri sayım mandalı)

"N olay olsun, sonra devam et." Bekleyenler sayaç sıfırlanana kadar durur. Tek kullanımlık.

CyclicBarrier (döngüsel bariyer)

"Hepimiz buluşma noktasına gelelim, sonra birlikte devam edelim." Yeniden kullanılabilir.

Exchanger (değiştirici)

İki iş parçacığı bir noktada buluşup **veri takası** yapar.

Semaphore (semafor)

N tane izin: aynı anda en çok N iş parçacığı kaynağa girebilir.

CountDownLatch

```
1 public static void main(String[] args)
2     throws Exception {
3     int sayac = 3;
4     ▶ CountDownLatch mandal = new CountDownLatch(sayac);
5     for (int i = 0; i < sayac; i++) {
6         int no = i + 1;    // lambda için kopya
7         new Thread(() -> {
8             System.out.println(no + ". olay gerçekleşti");
9             // Olay gerçekleştiğinde sayac azaltılır.
10            mandal.countDown();
11        }).start();
12    }
13    // iş parçacıklarının tamamlanması beklenir.
14    mandal.await();
15    System.out.println("Tüm olaylar tamamlandı!");
16 }
```

sayaç

3

1

2

3

olaylar

main

çalışıyor

Sayaç kadar olayı bekleyen iş parçacıklarını senkronize eder. **counter**: başlangıç sayısı, burada 3.

Adım 1 / 7

ÇIKTI

Tek kullanımlıktır: sayaç asla artmaz; sıfırken `countDown()` etkisizdir.

CountDownLatch

```
1 public static void main(String[] args)
2     throws Exception {
3     int sayac = 3;
4     CountDownLatch mandal = new CountDownLatch(sayac);
5     for (int i = 0; i < sayac; i++) {
6         int no = i + 1;    // lambda için kopya
7         new Thread(() -> {
8             System.out.println(no + ". olay gerçekleşti");
9             // Olay gerçekleştiğinde sayac azaltılır.
10            mandal.countDown();
11        }).start();
12    }
13    // iş parçacıklarının tamamlanması beklenir.
14    mandal.await();
15    System.out.println("Tüm olaylar tamamlandı!");
16 }
```

sayac

3

1

2

3

olaylar

main

çalışıyor

Üç iş parçacığı başlatılır; ne zaman çalışacakları belli değil.

Adım 2 / 7

ÇIKTI

Tek kullanımlıktır: sayac asla artmaz; sıfırken `countDown()` etkisizdir.

CountDownLatch

```
1 public static void main(String[] args)
2     throws Exception {
3     int sayac = 3;
4     CountDownLatch mandal = new CountDownLatch(sayac);
5     for (int i = 0; i < sayac; i++) {
6         int no = i + 1;    // lambda için kopya
7         new Thread(() -> {
8             System.out.println(no + ". olay gerçekleşti");
9             // Olay gerçekleştiğinde sayacı azaltılır.
10            mandal.countDown();
11        }).start();
12    }
13    // iş parçacıklarının tamamlanması beklenir.
14    ▶ mandal.await();
15    System.out.println("Tüm olaylar tamamlandı!");
16 }
```

sayacı

3

1

2

3

olaylar

main

WAITING (await)

await(): sayacı 0 olana kadar main bekler.

Adım 3 / 7

ÇIKTI

Tek kullanımlıdır: sayacı asla artmaz; sıfırken `countDown()` etkisizdir.

CountDownLatch

```
1 public static void main(String[] args)
2     throws Exception {
3     int sayac = 3;
4     CountDownLatch mandal = new CountDownLatch(sayac);
5     for (int i = 0; i < sayac; i++) {
6         int no = i + 1;    // lambda için kopya
7         new Thread(() -> {
8             ▶ System.out.println(no + ". olay gerçekleşti");
9             // Olay gerçekleştiğinde sayac azaltılır.
10            ▶ mandal.countDown();
11        }).start();
12    }
13    // iş parçacıklarının tamamlanması beklenir.
14    mandal.await();
15    System.out.println("Tüm olaylar tamamlandı!");
16 }
```

sayaç

2

1

2

3

olaylar

main

WAITING (await)

1. iş parçacığı yazar; **countDown()**
sayacı 1 azaltır: 2.

Adım 4 / 7

ÇIKTI

1. olay gerçekleşti

Tek kullanımlıktır: sayaç asla artmaz; sıfırken **countDown()** etkisizdir.

CountDownLatch

```
1 public static void main(String[] args)
2     throws Exception {
3     int sayac = 3;
4     CountDownLatch mandal = new CountDownLatch(sayac);
5     for (int i = 0; i < sayac; i++) {
6         int no = i + 1;    // lambda için kopya
7         new Thread(() -> {
8             ▶ System.out.println(no + ". olay gerçekleşti");
9             // Olay gerçekleştiğinde sayac azaltılır.
10            ▶ mandal.countDown();
11        }).start();
12    }
13    // iş parçacıklarının tamamlanması beklenir.
14    mandal.await();
15    System.out.println("Tüm olaylar tamamlandı!");
16 }
```

sayac

1

1

2

3

olaylar

main

WAITING (await)

Bu çalıştırmada sıradaki 3. iş parçacığıydı: sayac 1.

Adım 5 / 7

ÇIKTI

```
1. olay gerçekleşti
3. olay gerçekleşti
```

Tek kullanımlıktır: sayac asla artmaz; sıfırken `countDown()` etkisizdir.

CountDownLatch

```
1 public static void main(String[] args)
2     throws Exception {
3     int sayac = 3;
4     CountDownLatch mandal = new CountDownLatch(sayac);
5     for (int i = 0; i < sayac; i++) {
6         int no = i + 1;    // lambda için kopya
7         new Thread(() -> {
8             ▶ System.out.println(no + ". olay gerçekleşti");
9             // Olay gerçekleştiğinde sayac azaltılır.
10            ▶ mandal.countDown();
11        }).start();
12    }
13    // iş parçacıklarının tamamlanması beklenir.
14    mandal.await();
15    System.out.println("Tüm olaylar tamamlandı!");
16 }
```



main

devam

Son olay: sayac 0. Mandal açılır, main uyanır.

Adım 6 / 7

ÇIKTI

```
1. olay gerçekleşti
3. olay gerçekleşti
2. olay gerçekleşti
```

Tek kullanımlıktır: sayac asla artmaz; sıfırken `countDown()` etkisizdir.

CountDownLatch

```
1 public static void main(String[] args)
2     throws Exception {
3     int sayac = 3;
4     CountDownLatch mandal = new CountDownLatch(sayac);
5     for (int i = 0; i < sayac; i++) {
6         int no = i + 1;    // lambda için kopya
7         new Thread(() -> {
8             System.out.println(no + ". olay gerçekleşti");
9             // Olay gerçekleştiğinde sayac azaltılır.
10            mandal.countDown();
11        }).start();
12    }
13    // iş parçacıklarının tamamlanması beklenir.
14    mandal.await();
15    ▶ System.out.println("Tüm olaylar tamamlandı!");
16 }
```



main

devam

Bu satır her zaman en sonda yazılır; olay satırlarının sırası ise değişebilir.

Adım 7 / 7

ÇIKTI

```
1. olay gerçekleşti
3. olay gerçekleşti
2. olay gerçekleşti
Tüm olaylar tamamlandı!
```

Tek kullanımlıktır: sayac asla artmaz; sıfırken `countDown()` etkisizdir.

CyclicBarrier

```
public static void main(String[] args) {
    int parti = 3;
    Runnable eylem = () -> {
        System.out.println("Parti tamamlandı!");
    };
    CyclicBarrier bariyer =
        new CyclicBarrier(parti, eylem);
    for (int i = 0; i < parti; i++) {
        int no = i + 1;
        new Thread(() -> {
            try {
                System.out.println(no + " bekliyor");
                // iş parçacıklarını bekle
                bariyer.await();
                System.out.println(no + " devam ediyor");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

- Bir grup iş parçacığını **buluşma noktasında** (barrier) bekletir; **parti**: beklenecek sayı.
- `await()` : parti dolana kadar bekle. Dolunca **bariyer eylemi** bir kez çalışır, herkes devam eder.
- İstisna, zaman aşımı ya da kesme bariyeri **kırar**: bekleyenler `BrokenBarrierException` alır.

Çıktı (bir çalıştırma)

```
1 bekliyor
2 bekliyor
3 bekliyor
Parti tamamlandı!
3 devam ediyor
1 devam ediyor
2 devam ediyor
```

Exchanger

```
static void takas(Exchanger<String> ex, String veri, int ms) {  
    try {  
        System.out.println("gönderiliyor: " + veri);  
        Thread.sleep(ms);  
        String alinan = ex.exchange(veri);    // karşılıklı bekle  
        System.out.println(veri + " sahibi aldı: " + alinan);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}  
  
public static void main(String[] args) {  
    Exchanger<String> exchanger = new Exchanger<>();  
    // İlk iş parçasığı "Veri 1" gönderir, 2 sn sonra  
    new Thread(() -> takas(exchanger, "Veri 1", 2000)).start();  
    // İkinci iş parçasığı "Veri 2" gönderir, 1 sn sonra  
    new Thread(() -> takas(exchanger, "Veri 2", 1000)).start();  
}
```

Çıktı

```
gönderiliyor: Veri 2  
gönderiliyor: Veri 1  
Veri 1 sahibi aldı: Veri 2  
Veri 2 sahibi aldı: Veri 1
```

- İki iş parçasığının **veri alışverişi** için kullanılır; kilitlenmeden çok bir iletişim nesnesidir.
- `exchange(veri)`, karşı taraf da `exchange` çağırana kadar **bekler**: "Veri 2" sahibi 1 sn bekledi.
- Buluşunca her biri ötekinin verisini alır.

Semaphore

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

izinler



3 / 3



Semafor, eş zamanlı erişimi sınırlar: burada aynı anda en çok 3 iş parçacığı. `new Semaphore(n, true)` : adil dağıtım.

Adım 1 / 10

ÇIKTI



Semaphore

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

izinler



3 / 3

1	2	3	4	5
hazır	hazır	hazır	hazır	hazır

Beş iş parçacığı başlatılır. `acquire()` izin yoksa bekler, `release()` izni geri verir.

Adım 2 / 10

ÇIKTI



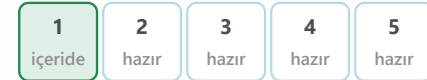
Semaphore

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

izinler



2 / 3



1 izin alır ve girer.

Adım 3 / 10

ÇIKTI

1 girdi

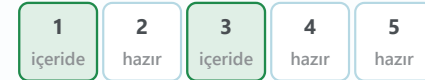
Semaphore

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

izinler



1 / 3



3 girer.

Adım 4 / 10

ÇIKTI

```
1 girdi  
3 girdi
```

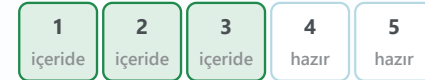
Semaphore

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

izinler



0 / 3



2 girer. İzin kalmadı.

Adım 5 / 10

ÇIKTI

```
1 girdi
3 girdi
2 girdi
```

Semaphore

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

izinler



0 / 3

4 ve 5 `acquire()` 'da bekler.

Adım 6 / 10

ÇIKTI

```
1 girdi
3 girdi
2 girdi
```

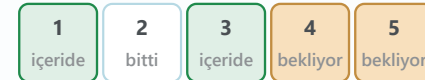
Semaphore

```
1 public static void main(String[] args) {
2     int izin = 3;
3     Semaphore semaphore = new Semaphore(izin);
4     for (int i = 0; i < 5; i++) {
5         int no = i + 1;
6         new Thread(() -> {
7             try {
8                 semaphore.acquire(); // izin al
9                 System.out.println(no + " girdi");
10                Thread.sleep(2000);
11                System.out.println(no + "
12                çıkıyor");
13                semaphore.release(); // izni
14                bırak
15            } catch (InterruptedException e) {
16                e.printStackTrace();
17            }
18        }).start();
19    }
20 }
```

izinler



1 / 3



2 sn sonra 2 çıkar ve iznini bırakır.

Adım 7 / 10

ÇIKTI

```
1 girdi
3 girdi
2 girdi
2 çıkıyor
```

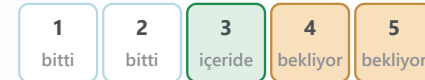
Semaphore

```
1 public static void main(String[] args) {
2     int izin = 3;
3     Semaphore semaphore = new Semaphore(izin);
4     for (int i = 0; i < 5; i++) {
5         int no = i + 1;
6         new Thread(() -> {
7             try {
8                 semaphore.acquire(); // izin al
9                 System.out.println(no + " girdi");
10                Thread.sleep(2000);
11                System.out.println(no + "
12                çıkıyor");
13                semaphore.release(); // izni
14                bırak
15            } catch (InterruptedException e) {
16                e.printStackTrace();
17            }
18        }).start();
19    }
20 }
```

izinler



2 / 3



1 de çıkar.

Adım 8 / 10

ÇIKTI

```
1 girdi
3 girdi
2 girdi
2 çıkıyor
1 çıkıyor
```

Semaphore

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

izinler



0 / 3



Bekleyen 4 ve 5 izinleri alıp girer.

Adım 9 / 10

ÇIKTI

```
1 girdi
3 girdi
2 girdi
2 çıkıyor
1 çıkıyor
4 girdi
5 girdi
```

Semaphore

```
1 public static void main(String[] args) {
2     int izin = 3;
3     Semaphore semaphore = new Semaphore(izin);
4     for (int i = 0; i < 5; i++) {
5         int no = i + 1;
6         new Thread(() -> {
7             try {
8                 semaphore.acquire(); // izin al
9                 System.out.println(no + " girdi");
10                Thread.sleep(2000);
11                System.out.println(no + "
12                çıkıyor");
13                semaphore.release(); // izni
14                bırak
15            } catch (InterruptedException e) {
16                e.printStackTrace();
17            }
18        }).start();
19    }
20 }
```

izinler



3 / 3

1	2	3	4	5
bitti	bitti	bitti	bitti	bitti

3, 4 ve 5 de çıkar; 3 izin geri döndü.

Adım 10 / 10

ÇIKTI

```
1 girdi
3 girdi
2 girdi
2 çıkıyor
1 çıkıyor
4 girdi
5 girdi
3 çıkıyor
4 çıkıyor
5 çıkıyor
```

Atomik Sınıflar

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

Çıktı

Sonuç: 200000
true 0
false 0

- `AtomicInteger`, `AtomicLong`, `AtomicBoolean` ...: kilitli, güvenli erişim.
- `incrementAndGet()` = atomik `++x` ; işlem yarıda kesilemez.
- Atomik nesne ≈ `volatile` alan + atomik `compareAndSet` .
- `compareAndSet(beklenen, yeni)` : değer hâlâ *beklenen* ise *yeni*'yi yazar, `true` döner; değilse dokunmaz.

compareAndSet ile Kilitsiz Artırma

```
► AtomicInteger sayac = new AtomicInteger(5);
2
3 int artir() {                                // incrementAndGet'in
    mantığı
4     while (true) {
5         int eski = sayac.get();
6         int yeni = eski + 1;
7         if (sayac.compareAndSet(eski, yeni))
8             return yeni;
9         // başarısız: araya biri girdi, tekrar dene
10    }
11 }
```

bellek

sayac
5

T1

eski = -
yeni = -

T2

eski = -
yeni = -

İki iş parçacığı aynı **sayac** nesnesini artıracak. Başlangıç değeri 5.

Adım 1 / 9

DEĞİŞKENLER

sayac = 5

ÇAĞRI YIĞINI

boş

compareAndSet ile Kilitsiz Artırma

```
1 AtomicInteger sayac = new AtomicInteger(5);
2
3 int artir() {                                // incrementAndGet'in
    mantığı
4     while (true) {
5         ▶ int eski = sayac.get();
6         int yeni = eski + 1;
7         if (sayac.compareAndSet(eski, yeni))
8             return yeni;
9         // başarısız: araya biri girdi, tekrar dene
10    }
11 }
```

bellek

sayac
5

T1

eski = 5
yeni = -

T2

eski = -
yeni = -

T1 güncel değeri okur: **eski** = 5.

Adım 2 / 9

DEĞİŞKENLER

T1.eski = 5

ÇAĞRI YIĞINI

T1: artir()

compareAndSet ile Kilitsiz Artırma

```
1 AtomicInteger sayac = new AtomicInteger(5);
2
3 int artir() {                                // incrementAndGet'in
  mantığı
4   while (true) {
5     ▶   int eski = sayac.get();
6     int yeni = eski + 1;
7     if (sayac.compareAndSet(eski, yeni))
8       return yeni;
9     // başarısız: araya biri girdi, tekrar dene
10  }
11 }
```

bellek

sayac
5

T1

eski = 5
yeni = -

T2

eski = 5
yeni = -

T1 devam etmeden T2 de okur: o da 5 görür.
Kilit yok, ikisi de çalışmaya devam eder.

Adım 3 / 9

DEĞİŞKENLER

T1.eski = 5

T2.eski = 5

ÇAĞRI YIĞINI

T2: artir()

T1: artir()

compareAndSet ile Kilitsiz Artırma

```
1 AtomicInteger sayac = new AtomicInteger(5);
2
3 int artir() {                                // incrementAndGet'in
    mantığı
4     while (true) {
5         int eski = sayac.get();
6         int yeni = eski + 1;
7         if (sayac.compareAndSet(eski, yeni))
8             return yeni;
9         // başarısız: araya biri girdi, tekrar dene
10    }
11 }
```

bellek

sayac
5

T1

eski = 5
yeni = 6

T2

eski = 5
yeni = 6

İkisi de yerel olarak **yeni** = 6 hesaplar. Sıradan **sayac++** olsaydı ikisi de 6 yazar, bir artış kaybolurdu.

Adım 4 / 9

DEĞİŞKENLER

T1.yeni = 6

T2.yeni = 6

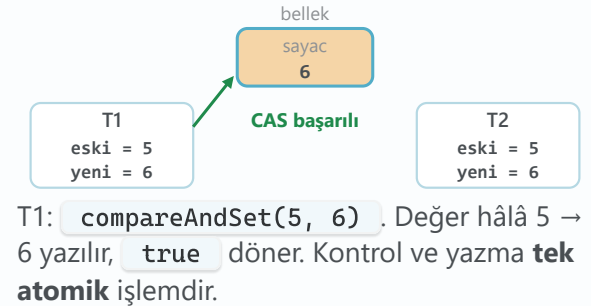
ÇAĞRI YIĞINI

T2: artir()

T1: artir()

compareAndSet ile Kilitsiz Artırma

```
1 AtomicInteger sayac = new AtomicInteger(5);
2
3 int artir() {                                // incrementAndGet'in
  mantığı
4   while (true) {
5       int eski = sayac.get();
6       int yeni = eski + 1;
7       if (sayac.compareAndSet(eski, yeni))
8           return yeni;
9       // başarısız: araya biri girdi, tekrar dene
10  }
11 }
```



Adım 5 / 9

DEĞİŞKENLER

sayac = 6

ÇAĞRI YIĞINI

T1: `artir()`

compareAndSet ile Kilitsiz Artırma

```
1 AtomicInteger sayac = new AtomicInteger(5);
2
3 int artir() {                                // incrementAndGet'in
  mantığı
4   while (true) {
5       int eski = sayac.get();
6       int yeni = eski + 1;
7       if (sayac.compareAndSet(eski, yeni))
8           return yeni;
9       // başarısız: araya biri girdi, tekrar dene
10  }
11 }
```



T2: `compareAndSet(5, 6)` . Değer artık 6, beklenen 5 değil → hiçbir şey yazılmaz, `false` .

Adım 6 / 9

DEĞİŞKENLER

sayac = 6

T2 CAS(5, 6) = false

ÇAĞRI YIĞINI

T2: artir()

compareAndSet ile Kilitsiz Artırma

```
1 AtomicInteger sayac = new AtomicInteger(5);
2
3 int artir() {                                // incrementAndGet'in
    mantığı
4     while (true) {
5         ▶ int eski = sayac.get();
6         int yeni = eski + 1;
7         if (sayac.compareAndSet(eski, yeni))
8             return yeni;
9         // başarısız: araya biri girdi, tekrar dene
10    }
11 }
```

bellek

sayac
6

T1

eski = -
yeni = -

T2

eski = 6
yeni = -

Döngü tekrar eder: T2 güncel değeri yeniden okur.

Adım 7 / 9

DEĞİŞKENLER

T2.eski = 6

ÇAĞRI YIĞINI

T2: artir()

compareAndSet ile Kilitsiz Artırma

```
1 AtomicInteger sayac = new AtomicInteger(5);
2
3 int artir() {                                // incrementAndGet'in
  mantığı
4   while (true) {
5     int eski = sayac.get();
6     int yeni = eski + 1;
7     if (sayac.compareAndSet(eski, yeni))
8       return yeni;
9     // başarısız: araya biri girdi, tekrar dene
10  }
11 }
```

bellek

sayac
6

T1

eski = -
yeni = -

yeni = 7.

T2

eski = 6
yeni = 7

Adım 8 / 9

DEĞİŞKENLER

T2.yeni = 7

ÇAĞRI YIĞINI

T2: artir()

compareAndSet ile Kilitless Artırma

```
1 AtomicInteger sayac = new AtomicInteger(5);
2
3 int artir() {                                // incrementAndGet'in
  mantığı
4   while (true) {
5       int eski = sayac.get();
6       int yeni = eski + 1;
7       if (sayac.compareAndSet(eski, yeni))
8           return yeni;
9       // başarısız: araya biri girdi, tekrar dene
10  }
11 }
```



Bu kez CAS başarılı: **sayac** = 7. İki artış da sayıldı; kimse kilit beklemedi, yalnızca T2 bir tur fazla döndü.

Adım 9 / 9

DEĞİŞKENLER

sayac = 7

ÇAĞRI YIĞINI

T2: artir()

Kesme (Interrupting)

```
public class KesmeOrnegi {  
    public static void main(String[] args)  
        throws InterruptedException {  
        Thread isci = new Thread() -> {  
            try {  
                while (!Thread.interrupted()) {  
                    // İş parçacığının çalışma mantığı  
                    System.out.println("çalışıyor...");  
                    Thread.sleep(400);  
                }  
            } catch (InterruptedException e) {  
                System.out.println("İş parçacığı iptal edildi!");  
            }  
        });  
        isci.start();  
        Thread.sleep(1000);  
        isci.interrupt(); // sleep() içindeyken kes  
        isci.join();  
        System.out.println("main: bitti");  
    }  
}
```

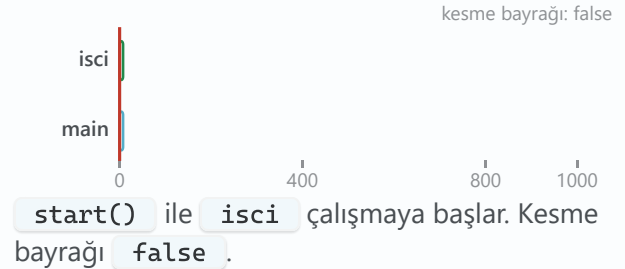
Çıktı

çalışıyor...
çalışıyor...
çalışıyor...
İş parçacığı iptal edildi!
main: bitti

- **interrupt()** : **kesme isteği**; iş parçacığı **sleep / wait** 'teyse **InterruptedException**.
- **Thread.interrupted()** bayrağı okur ve **temizler**; **isInterrupted()** yalnızca okur.

Kesme (interrupt) Adım Adım

```
► Thread isci = new Thread(() -> {  
2    try {  
3        while (!Thread.interrupted()) {  
4            System.out.println("çalışıyor...");  
5            Thread.sleep(400);  
6        }  
7    } catch (InterruptedException e) {  
8        System.out.println("iptal edildi!");  
9    }  
10 });  
► isci.start();  
12 Thread.sleep(1000);  
13 isci.interrupt(); // isci sleep() içinde  
14 isci.join();  
15 System.out.println("main: bitti");
```



Adım 1 / 9

DEĞİŞKENLER

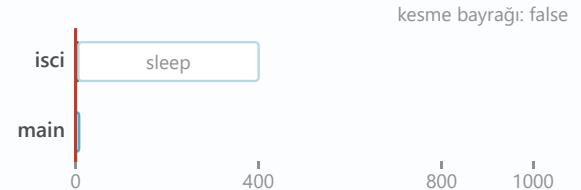
isc i = RUNNABLE

ÇIKTI



Kesme (interrupt) Adım Adım

```
1 Thread isci = new Thread(() -> {
2     try {
3         while (!Thread.interrupted()) {
4             System.out.println("çalışıyor...");
5             Thread.sleep(400);
6         }
7     } catch (InterruptedException e) {
8         System.out.println("iptal edildi!");
9     }
10 });
11 isci.start();
12 Thread.sleep(1000);
13 isci.interrupt(); // isci sleep() içinde
14 isci.join();
15 System.out.println("main: bitti");
```



Bayrak kapalı: döngüye girer, yazar ve 400 ms uyur.

Adım 2 / 9

DEĞİŞKENLER

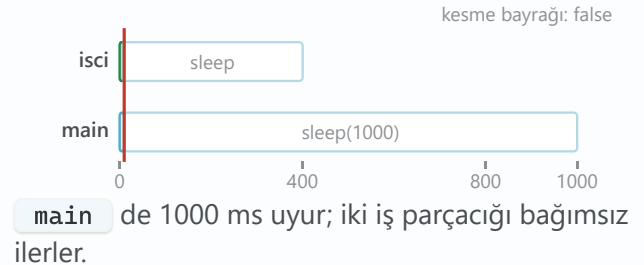
isci = TIMED_WAITING

ÇIKTI

çalışıyor...

Kesme (interrupt) Adım Adım

```
1 Thread isci = new Thread(() -> {
2     try {
3         while (!Thread.interrupted()) {
4             System.out.println("çalışıyor...");
5             Thread.sleep(400);
6         }
7     } catch (InterruptedException e) {
8         System.out.println("iptal edildi!");
9     }
10 });
11 isci.start();
12 Thread.sleep(1000);
13 isci.interrupt(); // isci sleep() içinde
14 isci.join();
15 System.out.println("main: bitti");
```



Adım 3 / 9

DEĞİŞKENLER

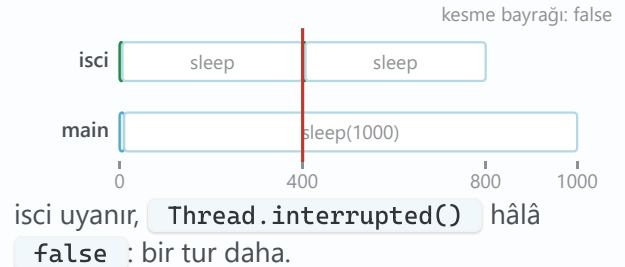
main = TIMED_WAITING

ÇIKTI

çalışıyor...

Kesme (interrupt) Adım Adım

```
1 Thread isci = new Thread(() -> {
2     try {
3         ▶ while (!Thread.interrupted()) {
4             ▶ System.out.println("çalışıyor...");
5             ▶ Thread.sleep(400);
6         }
7     } catch (InterruptedException e) {
8         System.out.println("iptal edildi!");
9     }
10 });
11 isci.start();
12 Thread.sleep(1000);
13 isci.interrupt(); // isci sleep() içinde
14 isci.join();
15 System.out.println("main: bitti");
```



Adım 4 / 9

DEĞİŞKENLER

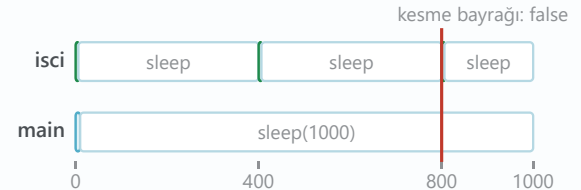
zaman = ≈ 400 ms

ÇIKTI

```
çalışıyor...
çalışıyor...
```

Kesme (interrupt) Adım Adım

```
1 Thread isci = new Thread(() -> {
2     try {
3         ▶ while (!Thread.interrupted()) {
4             ▶ System.out.println("çalışıyor...");
5             ▶ Thread.sleep(400);
6         }
7     } catch (InterruptedException e) {
8         System.out.println("iptal edildi!");
9     }
10 });
11 isci.start();
12 Thread.sleep(1000);
13 isci.interrupt(); // isci sleep() içinde
14 isci.join();
15 System.out.println("main: bitti");
```



Üçüncü tur: yazar ve yeniden uyur (1200 ms'ye kadar uyuyacaktı).

Adım 5 / 9

DEĞİŞKENLER

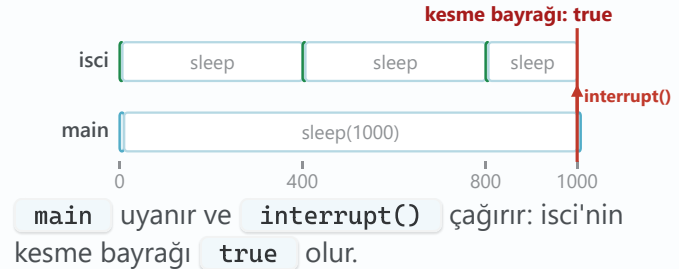
zaman = **≈800 ms**

ÇIKTI

```
çalışıyor...
çalışıyor...
çalışıyor...
```

Kesme (interrupt) Adım Adım

```
1 Thread isci = new Thread(() -> {
2     try {
3         while (!Thread.interrupted()) {
4             System.out.println("çalışıyor...");
5             Thread.sleep(400);
6         }
7     } catch (InterruptedException e) {
8         System.out.println("iptal edildi!");
9     }
10 });
11 isci.start();
12 Thread.sleep(1000);
13 ▶ isci.interrupt(); // isci sleep() içinde
14 isci.join();
15 System.out.println("main: bitti");
```



Adım 6 / 9

DEĞİŞKENLER

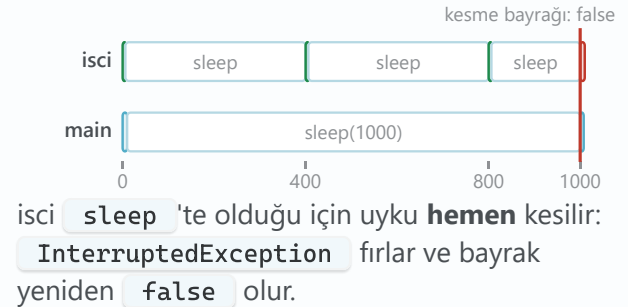
zaman = **≈1000 ms**

ÇIKTI

```
çalışıyor...
çalışıyor...
çalışıyor...
```

Kesme (interrupt) Adım Adım

```
1 Thread isci = new Thread(() -> {
2     try {
3         while (!Thread.interrupted()) {
4             System.out.println("çalışıyor...");
5             Thread.sleep(400);
6         }
7     } catch (InterruptedException e) {
8         System.out.println("iptal edildi!");
9     }
10 });
11 isci.start();
12 Thread.sleep(1000);
13 isci.interrupt(); // isci sleep() içinde
14 isci.join();
15 System.out.println("main: bitti");
```



Adım 7 / 9

DEĞİŞKENLER

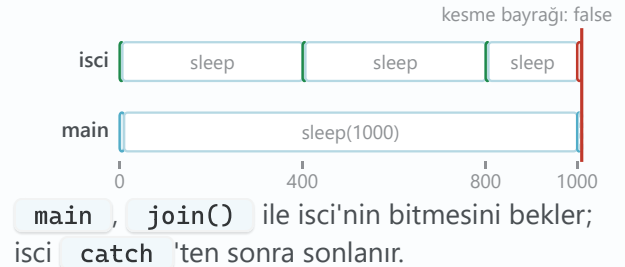
e = InterruptedException

ÇIKTI

```
çalışıyor...
çalışıyor...
çalışıyor...
iptal edildi!
```

Kesme (interrupt) Adım Adım

```
1 Thread isci = new Thread(() -> {
2     try {
3         while (!Thread.interrupted()) {
4             System.out.println("çalışıyor...");
5             Thread.sleep(400);
6         }
7     } catch (InterruptedException e) {
8         System.out.println("iptal edildi!");
9     }
10 });
11 isci.start();
12 Thread.sleep(1000);
13 isci.interrupt(); // isci sleep() içinde
14 ▶ isci.join();
15 System.out.println("main: bitti");
```



Adım 8 / 9

DEĞİŞKENLER

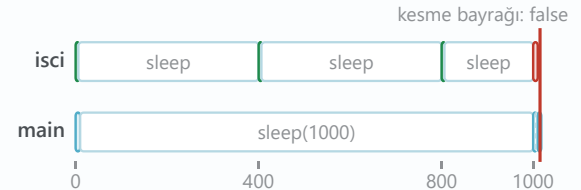
isci = TERMINATED

ÇIKTI

```
çalışıyor...
çalışıyor...
çalışıyor...
iptal edildi!
```

Kesme (interrupt) Adım Adım

```
1 Thread isci = new Thread(() -> {
2     try {
3         while (!Thread.interrupted()) {
4             System.out.println("çalışıyor...");
5             Thread.sleep(400);
6         }
7     } catch (InterruptedException e) {
8         System.out.println("iptal edildi!");
9     }
10 });
11 isci.start();
12 Thread.sleep(1000);
13 isci.interrupt(); // isci sleep() içinde
14 isci.join();
▶ System.out.println("main: bitti");
```



Program düzenli biter. Kesme bir **istektir**: iş parçacığı bayrağı ya da istisnayı kendisi ele alıp çıkar.

Adım 9 / 9

DEĞİŞKENLER

—

ÇIKTI

```
çalışıyor...
çalışıyor...
çalışıyor...
iptal edildi!
main: bitti
```

DiĞER ARAÇLAR

Askiya Alma (Suspension)

```
class Gorev implements Runnable {  
    private boolean duraklatildi = false;  
    public synchronized void duraklat() {  
        duraklatildi = true;  
    }  
    public synchronized void devamEt() {  
        duraklatildi = false;  
        notifyAll();    // bekleyeni uyandır  
    }  
    public void run() {  
        try {  
            for (int i = 1; i <= 5; i++) {  
                synchronized (this) {  
                    while (duraklatildi) wait();  
                }  
                System.out.println("adım " + i);  
                Thread.sleep(200);  
            }  
        } catch (InterruptedException e) { }  
    }  
}
```

```
Gorev g = new Gorev();  
Thread t = new Thread(g);  
t.start();  
Thread.sleep(500);  
g.duraklat();  
System.out.println("main: duraklatıldı");  
Thread.sleep(1000);  
System.out.println("main: devam");  
g.devamEt();  
t.join();
```

Çıktı

```
adım 1  
adım 2  
adım 3  
main: duraklatıldı  
main: devam  
adım 4  
adım 5
```

`suspend()` / `resume()` yerine
`wait()` / `notifyAll()` .

ThreadLocal Sınıfı

```
static final ThreadLocal<String> deger =  
    new ThreadLocal<>();  
public static void main(String[] args)  
    throws InterruptedException {  
    Thread thread1 = new Thread(() -> {  
        // İlk iş parçasığı  
        deger.set("Değer 1");  
        System.out.println("thread1: " + deger.get());  
    });  
    Thread thread2 = new Thread(() -> {  
        // İkinci iş parçasığı  
        deger.set("Değer 2");  
        System.out.println("thread2: " + deger.get());  
    });  
    thread1.start();  
    thread2.start();  
    thread1.join();  
    thread2.join();  
    System.out.println("main: " + deger.get());  
}
```

1. çalıştırma

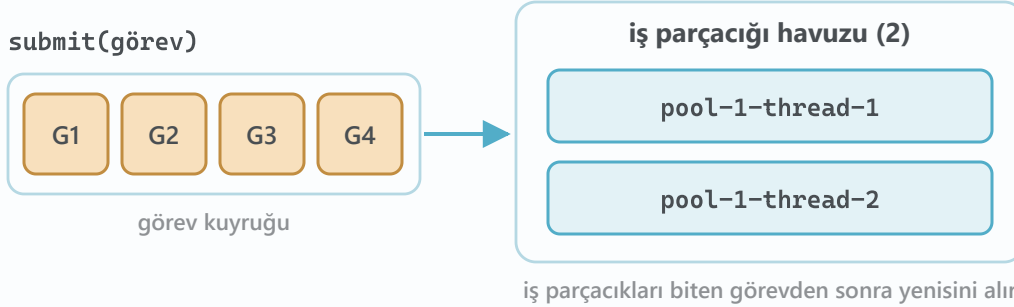
thread1: Değer 1
thread2: Değer 2
main: null

2. çalıştırma

thread2: Değer 2
thread1: Değer 1
main: null

- Her iş parçasığının **kendi ayrı kopyası** olan değişken; kopyalar birbirinden bağımsızdır.
- main hiç **set** etmediği için **null** görür.
- **ThreadLocal.withInitial(() -> 0)** (Java 8+) ya da **initialValue()** 'yu geçersiz kılmak: her iş parçasığına ayrı başlangıç değeri.

İş Parçacığı Havuzları (Thread Pools)



- Kullanılacak iş parçacıklarını **önceden oluşturup** yöneten yapı: görevler havuza gönderilir (submit).
- İş parçacığı oluşturmak pahalıdır; havuz iş parçacıklarını **yeniden kullanır** ve sayılarını sınırlar.
- `ExecutorService` arayüzü havuzdaki işleri kontrol eder;
`Executors.newFixedThreadPool(n)` hazır havuz verir.
- `ThreadPoolExecutor` : havuzu oluşturup yöneten, ayrıntılı ayarlanabilen sınıf.

DiĞER ARAÇLAR

ExecutorService

```
public class HavuzOrnegi {  
    static void gorev(int no) {  
        String ad = Thread.currentThread().getName();  
        System.out.println("Görev " + no + " -> " + ad);  
    }  
    public static void main(String[] args) throws Exception {  
        ExecutorService havuz = Executors.newFixedThreadPool(2);  
        for (int i = 1; i <= 4; i++) {  
            int no = i;  
            havuz.submit(() -> gorev(no));  
        }  
        // Callable: değer döndüren görev  
        Future<Integer> f = havuz.submit(() -> 6 * 7);  
        System.out.println("Callable sonucu: " + f.get());  
        havuz.shutdown(); // yeni görev kabul etme  
        havuz.awaitTermination(5, TimeUnit.SECONDS);  
        System.out.println("Havuz kapandı");  
    }  
}
```

Java 21+: `Executors.newVirtualThreadPerTaskExecutor()`
her göreve hafif bir **sanal iş parçacığı** (virtual thread) verir.

1. çalıştırma

Görev 1 -> pool-1-thread-1
Görev 2 -> pool-1-thread-2
Görev 3 -> pool-1-thread-2
Görev 4 -> pool-1-thread-2
Callable sonucu: 42
Havuz kapandı

2. çalıştırmada farklı olanlar

Görev 3 -> pool-1-thread-1
Görev 4 -> pool-1-thread-2

`Callable` değer
döndürür;

`Future.get()` sonucu
bekler. `shutdown()`
unutulursa havuzun iş
parçacıkları programı açık
tutar.

Havuz ve Future Adım Adım

```
1 static int is(int ms, int d) throws Exception {
2     Thread.sleep(ms);    // uzun süren iş yerine
3     return d;
4 }
5 // main içinde:
▶ ExecutorService havuz = Executors.newFixedThreadPool(2);
7 Future<Integer> f1 = havuz.submit(() -> is(300, 10));
8 Future<Integer> f2 = havuz.submit(() -> is(100, 20));
9 Future<Integer> f3 = havuz.submit(() -> is(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int toplam = f1.get() + f2.get() + f3.get();
12 System.out.println("toplam = " + toplam);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 havuz.shutdown();
```

kuyruk



işçi 1

işçi 2

0

100

200

300

main: çalışıyor

bitti: -

Sabit 2 iş parçacıklı bir havuz. İşçiler görev geldikçe çalışır, bitince yenisini alır.

Adım 1 / 10

DEĞİŞKENLER

havuz = 2 işçi

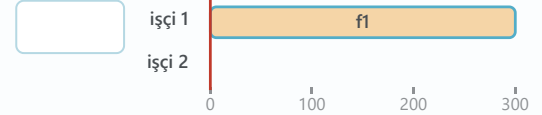
ÇIKTI

Süreler yaklaşıktır (iş parçacıklarının başlama gecikmesi hariç); görevlerin hangi işçiye düştüğü çalıştırmadan çalıştırmaya değişebilir.

Havuz ve Future Adım Adım

```
1 static int is(int ms, int d) throws Exception {
2     Thread.sleep(ms);    // uzun süren iş yerine
3     return d;
4 }
5 // main içinde:
6 ExecutorService havuz = Executors.newFixedThreadPool(2);
7 ▶ Future<Integer> f1 = havuz.submit(() -> is(300, 10));
8 Future<Integer> f2 = havuz.submit(() -> is(100, 20));
9 Future<Integer> f3 = havuz.submit(() -> is(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int toplam = f1.get() + f2.get() + f3.get();
12 System.out.println("toplam = " + toplam);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 havuz.shutdown();
```

kuyruk



main: çalışıyor

bitti: -

submit

görevi havuza verir ve hemen bir
Future döndürür. f1 işçi 1'de başlar.

Future

Adım 2 / 10

DEĞİŞKENLER

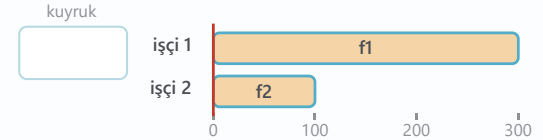
f1 = Future (bekliyor)

ÇIKTI

Süreler yaklaşıktır (iş parçacıklarının başlama gecikmesi hariç); görevlerin hangi işçiye düştüğü çalıştırmadan çalıştırmaya değişebilir.

Havuz ve Future Adım Adım

```
1 static int is(int ms, int d) throws Exception {
2     Thread.sleep(ms);    // uzun süren iş yerine
3     return d;
4 }
5 // main içinde:
6 ExecutorService havuz = Executors.newFixedThreadPool(2);
7 Future<Integer> f1 = havuz.submit(() -> is(300, 10));
8 ▶ Future<Integer> f2 = havuz.submit(() -> is(100, 20));
9 Future<Integer> f3 = havuz.submit(() -> is(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int toplam = f1.get() + f2.get() + f3.get();
12 System.out.println("toplam = " + toplam);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 havuz.shutdown();
```



main: çalışıyor

bitti: -

f2 boştaki işçi 2'ye gider.

Adım 3 / 10

DEĞİŞKENLER

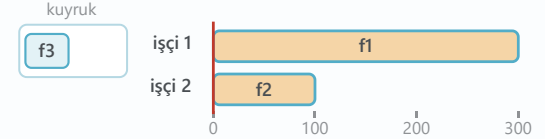
f2 = Future

ÇIKTI

Süreler yaklaşıktır (iş parçacıklarının başlama gecikmesi hariç); görevlerin hangi işçiye düştüğü çalıştırmadan çalıştırmaya değişebilir.

Havuz ve Future Adım Adım

```
1 static int is(int ms, int d) throws Exception {
2     Thread.sleep(ms);    // uzun süren iş yerine
3     return d;
4 }
5 // main içinde:
6 ExecutorService havuz = Executors.newFixedThreadPool(2);
7 Future<Integer> f1 = havuz.submit(() -> is(300, 10));
8 Future<Integer> f2 = havuz.submit(() -> is(100, 20));
9 ► Future<Integer> f3 = havuz.submit(() -> is(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int toplam = f1.get() + f2.get() + f3.get();
12 System.out.println("toplam = " + toplam);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 havuz.shutdown();
```



main: çalışıyor

bitti: -

İki işçi de meşgul: f3 **kuyrukta** bekler. Yeni iş parçacığı açılmaz.

Adım 4 / 10

DEĞİŞKENLER

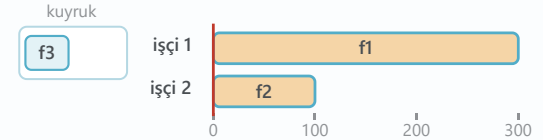
f3 = Future

ÇIKTI

Süreler yaklaşıktır (iş parçacıklarının başlama gecikmesi hariç); görevlerin hangi işçiye düştüğü çalıştırmadan çalıştırmaya değişebilir.

Havuz ve Future Adım Adım

```
1 static int is(int ms, int d) throws Exception {
2     Thread.sleep(ms);    // uzun süren iş yerine
3     return d;
4 }
5 // main içinde:
6 ExecutorService havuz = Executors.newFixedThreadPool(2);
7 Future<Integer> f1 = havuz.submit(() -> is(300, 10));
8 Future<Integer> f2 = havuz.submit(() -> is(100, 20));
9 Future<Integer> f3 = havuz.submit(() -> is(100, 30));
10 ▶ System.out.println(f1.isDone() + " " + f3.isDone());
11 int toplam = f1.get() + f2.get() + f3.get();
12 System.out.println("toplam = " + toplam);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 havuz.shutdown();
```



main: çalışıyor

bitti: -

Hiçbir görev bitmedi; `isDone()`
beklemeden yanıt verir.

Adım 5 / 10

DEĞİŞKENLER

`f1.isDone() = false`

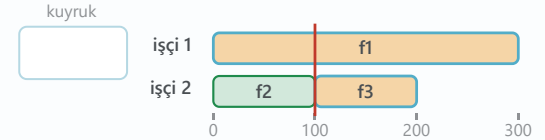
ÇIKTI

false false

Süreler yaklaşıktır (iş parçacıklarının başlama gecikmesi hariç); görevlerin hangi işçiye düştüğü çalıştırmadan çalıştırmaya değişebilir.

Havuz ve Future Adım Adım

```
1 static int is(int ms, int d) throws Exception {  
2     Thread.sleep(ms);    // uzun süren iş yerine  
3     return d;  
4 }  
5 // main içinde:  
6 ExecutorService havuz = Executors.newFixedThreadPool(2);  
7 Future<Integer> f1 = havuz.submit(() -> is(300, 10));  
8 Future<Integer> f2 = havuz.submit(() -> is(100, 20));  
9 Future<Integer> f3 = havuz.submit(() -> is(100, 30));  
10 System.out.println(f1.isDone() + " " + f3.isDone());  
▶ int toplam = f1.get() + f2.get() + f3.get();  
12 System.out.println("toplam = " + toplam);  
13 System.out.println(f2.isDone() + " " + f3.isDone());  
14 havuz.shutdown();
```



main: f1.get() bekliyor

bitti: f2

f1.get() sonuç hazır olana dek main'i **durdurur**. Bu sırada f2 biter, işçi 2 kuyruktan f3'ü alır.

Adım 6 / 10

DEĞİŞKENLER

zaman = ≈100 ms

ÇIKTI

false false

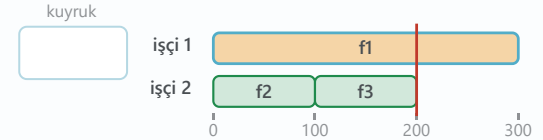
Süreler yaklaşıktır (iş parçacıklarının başlama gecikmesi hariç); görevlerin hangi işçiye düştüğü çalıştırmadan çalıştırmaya değişebilir.

Havuz ve Future Adım Adım

```

1 static int is(int ms, int d) throws Exception {
2     Thread.sleep(ms);    // uzun süren iş yerine
3     return d;
4 }
5 // main içinde:
6 ExecutorService havuz = Executors.newFixedThreadPool(2);
7 Future<Integer> f1 = havuz.submit(() -> is(300, 10));
8 Future<Integer> f2 = havuz.submit(() -> is(100, 20));
9 Future<Integer> f3 = havuz.submit(() -> is(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 ► int toplam = f1.get() + f2.get() + f3.get();
12 System.out.println("toplam = " + toplam);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 havuz.shutdown();

```



main: f1.get() bekliyor

bitti: f2, f3

f3 de biter; **main** hâlâ f1'i bekliyor.

Adım 7 / 10

DEĞİŞKENLER

zaman = **≈200 ms**

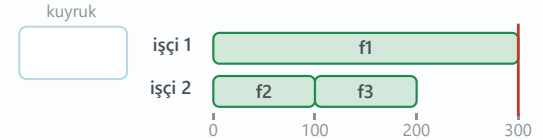
ÇIKTI

false false

Süreler yaklaşıktır (iş parçacıklarının başlama gecikmesi hariç); görevlerin hangi işçiye düştüğü çalıştırmadan çalıştırmaya değişebilir.

Havuz ve Future Adım Adım

```
1 static int is(int ms, int d) throws Exception {  
2     Thread.sleep(ms);    // uzun süren iş yerine  
3     return d;  
4 }  
5 // main içinde:  
6 ExecutorService havuz = Executors.newFixedThreadPool(2);  
7 Future<Integer> f1 = havuz.submit(() -> is(300, 10));  
8 Future<Integer> f2 = havuz.submit(() -> is(100, 20));  
9 Future<Integer> f3 = havuz.submit(() -> is(100, 30));  
10 System.out.println(f1.isDone() + " " + f3.isDone());  
▶ int toplam = f1.get() + f2.get() + f3.get();  
12 System.out.println("toplam = " + toplam);  
13 System.out.println(f2.isDone() + " " + f3.isDone());  
14 havuz.shutdown();
```



main: çalışıyor

bitti: f1, f2, f3

$t \approx 300$ ms: f1 hazır. f2 ve f3 zaten bittiği için `get()` 'leri beklemeden döner. Toplam süre 500 değil ≈ 300 ms.

Adım 8 / 10

DEĞİŞKENLER

toplam = 10+20+30=60

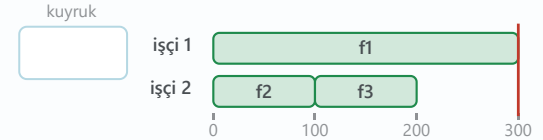
ÇIKTI

false false

Süreler yaklaşıktır (iş parçacıklarının başlama gecikmesi hariç); görevlerin hangi işçiye düştüğü çalıştırmadan çalıştırmaya değişebilir.

Havuz ve Future Adım Adım

```
1 static int is(int ms, int d) throws Exception {
2     Thread.sleep(ms);    // uzun süren iş yerine
3     return d;
4 }
5 // main içinde:
6 ExecutorService havuz = Executors.newFixedThreadPool(2);
7 Future<Integer> f1 = havuz.submit(() -> is(300, 10));
8 Future<Integer> f2 = havuz.submit(() -> is(100, 20));
9 Future<Integer> f3 = havuz.submit(() -> is(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int toplam = f1.get() + f2.get() + f3.get();
12 System.out.println("toplam = " + toplam);
13 System.out.println(f2.isDone() + " " + f3.isDone());
14 havuz.shutdown();
```



main: çalışıyor

bitti: f1, f2, f3

Sonuç ve durumlar yazılır.

Adım 9 / 10

DEĞİŞKENLER

toplam = 60

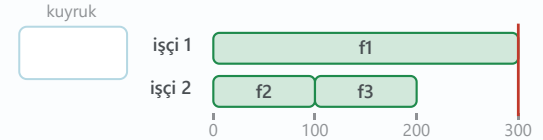
ÇIKTI

```
false false
toplam = 60
true true
```

Süreler yaklaşıktır (iş parçacıklarının başlama gecikmesi hariç); görevlerin hangi işçiye düştüğü çalıştırmadan çalıştırmaya değişebilir.

Havuz ve Future Adım Adım

```
1 static int is(int ms, int d) throws Exception {
2     Thread.sleep(ms);    // uzun süren iş yerine
3     return d;
4 }
5 // main içinde:
6 ExecutorService havuz = Executors.newFixedThreadPool(2);
7 Future<Integer> f1 = havuz.submit(() -> is(300, 10));
8 Future<Integer> f2 = havuz.submit(() -> is(100, 20));
9 Future<Integer> f3 = havuz.submit(() -> is(100, 30));
10 System.out.println(f1.isDone() + " " + f3.isDone());
11 int toplam = f1.get() + f2.get() + f3.get();
12 System.out.println("toplam = " + toplam);
13 System.out.println(f2.isDone() + " " + f3.isDone());
▶ havuz.shutdown();
```



main: bitti

bitti: f1, f2, f3

shutdown() yeni görev almayı bırakır; işçiler sonlanır. Unutulursa program kapanmaz.

Adım 10 / 10

DEĞİŞKENLER

havuz = kapanıyor

ÇIKTI

```
false false
toplam = 60
true true
```

Süreler yaklaşıktır (iş parçacıklarının başlama gecikmesi hariç); görevlerin hangi işçiye düştüğü çalıştırmadan çalıştırmaya değişebilir.

DiĞER ARAÇLAR

CompletableFuture, Eş Zamanlı Koleksiyonlar, 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> say =
    new ConcurrentHashMap<>();
String[] k = {"elma", "armut", "elma"};
Arrays.stream(k).parallel()
    .forEach(w -> say.merge(w, 1, Integer::sum));
System.out.println(say); // {elma=2, armut=1}
```

- **CompletableFuture** : **get()** ile beklemek yerine işlemleri **zincirler**; her adım önceki bitince arka planda çalışır.
- **ConcurrentHashMap** : parçalı kilitleme ile güvenli eşlem; **merge**, **compute** atomiktir.
- **CopyOnWriteArrayList** : her yazmada kopyalar; çok okunan, az yazılan listeler için.
- **ForkJoinPool** : böl ve yönet görevleri (**RecursiveTask**); boştaki işçi başkasının kuyruğundan iş **çalar**. Paralel akışlar (**parallelStream()**) bu havuzu kullanır.
- Paralel akış her zaman hızlı değildir: küçük veri ve paylaşılan değişken durum yavaşlatır ya da bozar. Önce ölçün.

İş Parçacığı Güvenliği (Thread Safety)

Bir sınıf, birden çok iş parçacığından aynı anda kullanıldığında da doğru davranıyorsa **iş parçacığı güvenlidir**.

Değiştirilemez nesneler

Oluşturulduktan sonra değişmeyen nesneler (`String` , `final` alanlı sınıflar, Java 16+ `record`) senkronizasyonsuz paylaşılabilir.

Paylaşmamak

Yerel değişkenler her iş parçacığının yığınındadır; `ThreadLocal` da kopya verir. Paylaşılmayan veri yarışmaz.

Senkronizasyon

Paylaşılan değişen veriye her erişim aynı kilitte korunur: `synchronized` , `Lock` .

Hazır yapılar

`AtomicInteger` , `ConcurrentHashMap` , `CopyOnWriteArrayList` , `BlockingQueue` ; `ArrayList` ve `HashMap` güvenli değildir.

ÖZET

Özet

Konu	Ana fikir	Araç
İş parçacığı	Süreç içinde bağımsız yürütme akışı; heap ortak, yığın ayrı	<code>Thread</code> , <code>Runnable</code>
Başlatma	İş <code>run()</code> 'da; başlatmak için <code>start()</code>	<code>start()</code> , <code>join()</code>
Yaşam döngüsü	NEW → RUNNABLE ⇌ BLOCKED / WAITING / TIMED_WAITING → TERMINATED	<code>getState()</code>
Durdurma	İş birliğiyle: bayrak ya da kesme	<code>volatile</code> , <code>interrupt()</code>
Yarış durumu	Paylaşılan veride bölünebilir işlemler; sonuç sıraya bağlı	<code>synchronized</code> , <code>Lock</code>
Görünürlük	Bir iş parçacığının yazdığı diğerine görünmeyebilir	<code>volatile</code>
Kilitlenme	Kilitleri farklı sırada almak döngü yaratır	Sabit sıra, <code>tryLock</code>
Koşul bekleme	Kilidi bırakarak bekle, değişince uyandır	<code>wait/notify</code> , <code>Condition</code>
Koordinasyon	Sayma, buluşma, takas, izin	<code>CountDownLatch</code> , <code>CyclicBarrier</code> , <code>Exchanger</code> , <code>Semaphore</code>
Diğer	Kilitsiz sayaç, iş parçacığına özel değer, havuz	<code>AtomicInteger</code> , <code>ThreadLocal</code> , <code>ExecutorService</code>

İlgili Sayfalar

Bölüm 9 Çalışma Soruları

İş parçacığı durumları, yarış durumu, senkronizasyon, kilitlenme ve iş parçacığı havuzları üzerine çözümlü sorular.

Terim sözlüğü

İş parçacığı, eş zamanlılık, kilitlenme, monitör, atomik işlem, iş parçacığı havuzu gibi terimlerin Türkçe ve İngilizce karşılıkları.

Sık Yapılan Hatalar (1)

Hata	Belirti	Doğrusu
<code>t.run()</code> çağırmak	Yeni iş parçacığı yok; kod main'de sırayla çalışır	<code>t.start()</code>
<code>start()</code> 'ı iki kez çağırmak	<code>IllegalThreadStateException</code>	Her çalıştırma için yeni <code>Thread</code>
<code>Thread.sleep(100);</code> yalnız başına	<code>error: unreported exception InterruptedException; must be caught or declared to be thrown</code>	<code>try-catch</code> ya da <code>throws InterruptedException</code>
Lambda içinde döngü değişkeni <code>i</code>	<code>error: local variables referenced from a lambda expression must be final or effectively final</code>	<code>int no = i;</code> kopyası
Fırlatılmayan istisnayı yakalamak: <code>try { while (!Thread.interrupted()) { } } catch (InterruptedException e)</code>	<code>error: exception InterruptedException is never thrown in body of corresponding try statement</code>	Gövdede <code>sleep / wait</code> olmalı ya da <code>catch</code> kaldırılmalı
Sınıf <code>Thread</code> 'i genişletmiyorken <code>currentThread()</code>	<code>error: cannot find symbol</code>	<code>Thread.currentThread()</code>
<code>join()</code> 'u unutmak	main, sonuç hazır olmadan okur (ör. 0)	Sonucu okumadan önce <code>join()</code>

Sık Yapılan Hatalar (2)

Hata	Belirti	Doğrusu
<code>wait()</code> / <code>notify()</code> 'ı <code>synchronized</code> dışında çağırmak	<code>IllegalMonitorStateException: current thread is not owner</code>	Aynı nesne üzerinde <code>synchronized</code> içinde
Koşulu <code>if</code> ile beklemek	Uyanınca koşul yine bozuk olabilir; taşan/boş kuyruk	<code>while (koşul) wait();</code>
<code>unlock()</code> 'u <code>finally</code> dışında yazmak	İstisnada kilit hiç açılmaz; herkes sonsuza dek bekler	<code>lock(); try { ... } finally { unlock(); }</code>
Almadığı kilidi açmak	<code>IllegalMonitorStateException</code>	Her <code>lock()</code> için bir <code>unlock()</code>
<code>volatile</code> sayaçta <code>sayac++</code>	Artışlar kaybolur (ör. 163299)	<code>AtomicInteger</code> ya da <code>synchronized</code>
<code>final volatile</code> alan	<code>error: illegal combination of modifiers: final and volatile</code>	Birini seçin
<code>stop()</code> , <code>suspend()</code> kullanmak	Java 20+: <code>UnsupportedOperationException</code>	Bayrak, <code>interrupt()</code> , <code>wait/notify</code>
<code>InterruptedException</code> 'ı boş <code>catch</code> ile yutmak	Kesme isteği kaybolur, iş parçacığı durmaz	Döngüden çık ya da <code>Thread.currentThread().interrupt()</code>
Kilitleri farklı sırada almak	Kilitlenme: program donar	Herkes aynı sırayla alsın

PEKİŞTİRME

Örnek Problemler (1)

1) ÇIKTI NEDİR? HER ÇALIŞTIRMADA AYNI MI?

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

2) ÇIKTI NEDİR?

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

3) AŞAĞIDAKİLERDEN HANGİLERİ "İKİ İŞ PARÇACIĞI" ÖRNEĞİNİN OLASI ÇIKTISIDIR?

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

► Cevaplar

PEKİŞTİRME

Örnek Problemler (2)

4) HER DURUMDA İŞ PARÇACIĞININ DURUMU NE?

a) `sleep(500)` içinde · b) dolu bir `synchronized` bloğa girerken · c) `wait()` sonrası · d) `run()` bitince

5) İŞ PARÇACIĞI GÜVENLİ Mİ?

```
class Hesap {  
    private int bakiye = 0;  
    void yatir(int m) { bakiye += m; }  
    int getBakiye() { return bakiye; }  
}
```

6) İKİ İŞ PARÇACIĞI BUNLARI ÇALIŞTIRIYOR. NE OLABİLİR?

```
// T1  
synchronized (a) {  
    synchronized (b) { }  
}  
  
// T2  
synchronized (b) {  
    synchronized (a) { }  
}
```

► Cevaplar

Alıştırmalar (1)

- **1.** 1..1000 sayılarının toplamını iki iş parçacığıyla hesaplayın: biri ilk yarıyı, öteki ikinci yarıyı tolasın; main ikisini bekleyip toplamı yazsın.
- **2.** 1..1000 arasındaki çift sayıları dört iş parçacığıyla (her biri 250 sayı) ortak bir sayaçta sayın. Sayaç doğru çıkmalı.

► Cevaplar

PEKİŞTİRME

Alıştırmalar (2)

- 3. 1..5 sayılarının karelerini 3 iş parçacıklı bir havuzda `Callable` görevleriyle hesaplayın ve `Future` 'lardan toplayın.
- 4. Her saniye "tik" yazan bir iş parçacığı yazın; main 3,5 sn sonra onu `interrupt()` ile durdursun.

► Cevaplar

Kaynaklar

- Oracle, *The Java Tutorials*: Lesson: Concurrency (Processes and Threads, Synchronization, Liveness, Guarded Blocks, High Level Concurrency Objects).
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21 Edition, Bölüm 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 ve diğ., *Java Concurrency in Practice*, Addison-Wesley, 2006.
- P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 23: Concurrency).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson (Bölüm 12: Concurrency).
- K. Sierra, B. Bates ve T. Gee, *Head First Java*, 3. baskı, O'Reilly (iş parçacıkları ve eş zamanlılık bölümleri).
- J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley (Bölüm 11: Concurrency, Madde 78–84).

Son

Bölüm 9: İş Parçacıkları

Sonraki bölüm: Bölüm 10: Veri Tabanı · [Terim sözlüğü](#)