Bölüm 9: İş Parçacıkları

JAVA ile Nesne Yönelimli Programlama

İş Parçacığı (Thread)

- Bir Java programındaki temel yürütme birimidir.
- Her iş parçacığı, bir Thread sınıfı nesnesi tarafından temsil edilir.
- İş parçacıkları, çoklu görev (multithreading) ve eşzamanlı programlama için kullanılır.

Java Sanal Makinesi (JVM)

- Java programlarının çalıştığı sanal bir ortamdır.
- JVM, iş parçacıklarını bağımsız yürütme birimleri olarak yönetir.

Thread Sınıfı

- Her iş parçacığı, java.lang paketinde bulunan Thread sınıfından türetilir.
- İş parçacığının yaşam döngüsünü kontrol ve yönetmek için kullanılır.
- İş parçacığı yönetimi için gerekli olan metodları ve özellikleri içerir.
- İş parçacığı oluşturmak için Thread sınıfından türeyen alt sınıf oluşturulur.
  - run() metodu override edilerek program ana mantığı tanımlanır.
  - start() metodu çağrılarak iş parçacığı başlatılır.

İş Parçacığı Yönetimi

- **sleep**(): İş parçacığını belirli bir süreyle uyutur.
- **join**(): Bir iş parçacığının tamamlanmasını bekler.
- **yield**(): İş parçacığının kontrolü geçici olarak bırakmasını sağlar.

İş Parçacığının Temel Nitelikleri

- **Runnable** (Çalıştırılabilir):
  - run() metodunu içeren nesne.
  - İş parçacığının ana mantığı burada tanımlanır.
- **Name** (İsim):
  - İş parçacığının adı.
  - Günlük kayıtları (log) veya teşhis amaçlı kullanılır.
- **ID** (Kimlik):
  - İş parçacığının tekil kimliği.
  - Sistem tarafından iş parçacığı oluşturulduğunda otomatik olarak atanır.

İş Parçacığının Temel Nitelikleri

- **ThreadGroup** (İş Parçacığı Grubu):
  - İş parçacığının ait olduğu grup.
  - Grup, iş parçacıklarını düzenlemek ve yönetmek için kullanılır.
- **Daemon** (Arka Plan İş Parçacığı):
  - Daemon iş parçacıkları, diğer iş parçacıkları için hizmet sağlar
  - Belirli görevleri periyodik gerçekleştirir. Tamamlanması beklenmez.
- **ContextClassLoader** (Bağlam Sınıf Yükleyici):
  - İş parçacığı tarafından kullanılan sınıf yükleyici.
  - Dinamik olarak sınıfları yüklemek için kullanılır.

İş Parçacığının Temel Nitelikleri

- **Priority** (Öncelik):
  - Thread.MIN_PRIORITY ve Thread.MAX_PRIORITY arasında tamsayı.
  - İş parçacığının öncelik düzeyini belirler.
- **State** (Durum):
  - İş parçacığının mevcut durumu.
  - Örnek durumlar: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED.
- **Interrupted** (Kesilme Durumu):
  - İş parçacığının kesilip kesilmediği durumu.
  - interrupt() metodu kullanılarak işaretlenir.

JVM Başlangıcında İki İş Parçacığı Grubu

- **Main Thread Group** (Ana İş Parçacığı Grubu):
  - Ana iş parçacığı ve oluşturulan iş parçacıkları bu gruba aittir.
  - JVM başladığında varsayılan olarak oluşturulur.
- **System Thread Group** (Sistem İş Parçacığı Grubu):
  - JVM tarafından oluşturulan Garbage Collector, Signal Dispatcher ve Finalizer iş parçacıkları bu gruba dahildir.

JVM Başlangıcında Dört Temel İş Parçacığı

- **Main Thread** (Ana İş Parçacığı):
  - JVM başladığında ilk çalışan iş parçacığıdır.
  - main() metodu burada çalıştırılır.
  - Temel uygulama mantığı burada yürütülür.
- **Garbage Collector Thread** (Çöp Toplayıcı İş Parçacığı):
  - Bellek yönetiminden sorumludur.
  - Referansı olmayan nesneleri bulup temizler.
  - Otomatik çalışır.

JVM Başlangıcında Dört Temel İş Parçacığı

- **Signal Dispatcher Thread** (Sinyal Gönderici İş Parçacığı):
  - İşletim sisteminden gelen sinyalleri yönetir.
  - Örneğin, SIGSEGV (Segmentation Fault) gibi hata sinyallerini ele alır.
  - JVM'yi güvenli kapatma işlemlerini yönetir.
- **Finalizer Thread** (Sonlandırıcı İş Parçacığı):
  - Nesnelerin finalize() metodunu çağırır.
  - Modern Java uygulamalarında pek önerilmez, çünkü try-finally veya AutoCloseable kullanımı tercih edilir.

İş Parçacığı Durumları

- İş parçacıkları yaşam döngülerinde farklı durumlara sahiptir.
- NEW (Yeni):
  - İş parçacığı henüz başlatılmamıştır.
  - start() metodunu çağırarak başlatılır.
- RUNNABLE (Çalışabilir):
  - İş parçacığı çalışıyor, ancak işlemci gibi kaynaklara ihtiyaç duyabilir.
  - Çalışan veya bekleyen durumda olabilir.

İş Parçacığı Durumları

- BLOCKED (Engellenmiş):
  - İş parçacığı, synchronized blok/metoda giriş yapmak için bir monitör kilidi bekliyor.
  - Diğer bir iş parçacığı bu kilit üzerinde işlem yapana kadar bekler.
- WAITING (Bekliyor):
  - İş parçacığı, şu metodlardan birini çağırarak bekliyor:
    - Object.wait (süresiz)
    - Thread.join (süresiz)
    - LockSupport.park

İş Parçacığı Durumları

- TIMED_WAITING (Zamanlı bekleme):
  - İş parçacığı, bekleme süresi ile şu metotlardan birini çağırarak bekliyor:
    - Thread.sleep
    - Object.wait (belirli bir süreyle)
    - Thread.join (belirli bir süreyle)
    - LockSupport.parkNanos
    - LockSupport.parkUntil
- TERMINATED (Sonlandırılmış):
  - İş parçacığı, yürütmesini tamamlamıştır.
  - run() metodunun işi tamamlandığında bu duruma geçer.

Runnable Arayüzü

- Her iş parçacığı, çalıştığı bir "runnable" nesneye sahiptir.
- Runnable arayüzü, yalnızca run() metodunu içerir.
- Runnable arayüzü gerçeklenerek iş parçacığının nesnesini oluşturulur.

```
package java.lang;

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

Runnable Arayüzü

- Runnable arayüzünü gerçekleyen sınıf, run() metodunu tanımlamak zorundadır.
- İş parçacığı, run() metodu çağırılarak çalıştırılabilir.

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

İş Parçacığını Başlatma

- İş parçacığını başlatmak için Thread sınıfının yapıcı metoduna parametre olarak çalıştırılabilir bir nesne verilir.
- start() metodu çağrılarak iş parçacığı başlatılır.
- run() metodu doğrudan çağrılmaz, aksi takdirde iş parçacığı çağıranın iş parçacığında çalışır.

İş Parçacığını Başlatma

```
public class Worker implements Runnable {
  // ... İş parçacığının görevleri burada tanımlanır
  public void run() {
    // ...
  }

  public static void main(String[] args) {
    //İş parçacığını başlatma
    Worker w = new Worker();
    Thread t = new Thread(w);
    t.start();
  }
}
```

İş Parçacığını Başlatma

- Thread sınıfı genişletilerek de iş parçacığı oluşturulabilir.
- Runnable arayüzünü gerçeklemek yerine Thread sınıfı kullanılabilir.

```
public class Worker extends Thread {
  // İş parçacığının görevleri burada tanımlanır
  public void run() {
    // ...
  }

  public static void main(String[] args) {
    //İş parçacığını oluşturma ve başlatma
    Worker w = new Worker();
    w.start();
  }
}
```

Tek Miras Sorunu

- Java'da tek bir sınıftan kalıtım mümkündür.
- Çalıştırılabilir nesne bir sınıftan türemişse, Thread sınıfını genişletemez.
- Nesne içerisinde Thread nesnesi oluşturularak bu sınırlama aşılabilir.

Tek Miras Sorunu

```
public class Worker extends Ata implements Runnable {
  // İş parçacığının görevleri burada tanımlanır
  private Thread thread;

  public void run() {
    // ...
  }

  public void start() {
    thread = new Thread(this);
    thread.start();
  }
}
```

İş Parçacığını Durdurma

- İş parçacığı, run metodunu tamamladığında çalışmasını durdurur.
- JVM,
  - Runtime.exit() çağrıldığında
  - Tüm daemon olmayan iş parçacıkları sonlandığında kapanır.
- İş parçacığını zorla durdurmak için, periyodik olarak bir bayrak (flag) kontrol edilir.
- Bayrak, iş parçacıklarının değişiklikleri görmesi için volatile tanımlanır.
- *Don't call deprecated Thread.stop()*

İş Parçacığını Durdurma

```
public class Worker extends Thread {

   private volatile boolean flag = false;

   public void finish() {flag = true;}

   public void run() {
      while (!flag) {
         // ...
      }
   }
   //...
}
```

İş Parçacığını Durdurma

```
public class Worker implements Runnable {

  private volatile Thread thread = new Thread(this);

  public void finish() {thread = null;}

  public void run() {
    while (thread == currentThread()) {
      // ...
    }
  }
  // ...
}
```

İşlemleri Doğru Sırayla Gerçekleştirmek

- Her iş parçacığının çalışma hızı ve sıklığı değişkenlik gösterir.
- İşletim sistemleri, öncelikleri görmezden gelebilir.
- Bir iş parçacığı bellekten bir değişkeni okuduktan sonra, başka bir iş parçacığı değişkenin değerini güncelleyebilir.
- İş parçacığı güncel olmayan değerlerle çalışıyor olabilir.
- Derleyiciler, optimizasyon yaparken iş parçacığı içindeki ifadeleri yeniden düzenleyebilir, bu da beklenmeyen davranışlara neden olabilir.

Senkronizasyon Seçenekleri

- join() metodu
- synchronized ifadesi
- Lock nesneleri
- Barriers, semaphores, ve exchangers
- Volatile değişkenler
- Atomic nesneler
- *priorities, sleeping, yielding, and timers look like synchronization but not.*

synchronized İfadesi

- Her nesne, kilitlenip kilidini açabileceğiniz bir monitöre sahiptir.
- Monitör, aynı anda yalnızca bir iş parçacığı tarafından sahip olunabilir.
- t1 iş parçacığı monitöre sahipse, t2 monitöre sahip olmak isterse, t2 bekler.
- Monitör serbest bırakıldığında, bekleyen iş parçacıkları rekabet eder ve sadece biri sahip olabilir.
- Monitörü kilitlemek veya kilidini açmak için synchronized ifadesi kullanılır.
- Metotları senkronize tanımlamak, metodun içindeki kod bloğunu senkronize etmekle aynıdır.
- Senkronizasyon, karşılıklı dışlamayı sağlayarak veri bütünlüğünü korur.

Monitör

- Bir nesnenin monitörüne synchronized ile sahip olunur.

```
synchronized (o) {
// ...
}
```

- Kilidin serbest bırakılması, ifadenin sonunda gerçekleşir.

Metodları Senkronize Tanımlama

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

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

Karşılıklı Dışlama

```
class OncelikliKuyruk {
  private Object[] veri;
  private int boyut;

  public synchronized ekle(Object o) {
    if (boyut == veri.length) {
      throw new HeapFullException();
    }
    veri[boyut++] = o;
  }

  public synchronized Object cikar() { ... }
  public synchronized String toString() { ... }
}
```

volatile Nitelikler

- Paylaşılan veri tek bir değişken ise, nesneyi kilitlemeye gerek olmadan karşılıklı dışlama sağlanır.
- Java, değişkenlerin (long ve double hariç) okuma ve yazma işlemlerinin atomik olduğunu garanti eder.
- *volatile can let you avoid synchronized statement and associated lock.*
- Volatile kelimesi, bir değişkenin değerinin ana bellekten okunmasını ve değiştiğinde tekrar ana belleğe yazılmasını garanti eder.
- Ana bellek yerine, performans amaçlı yazmaç (register) kullanıldığında, diğer iş parçacıkları güncel değere erişemezler.
- *Note only loads and stores are atomic, an expression like x++ is not.*

final volatile Birlikte Kullanımı

- *What happens if you make a field both final and volatile?*
- final değişkenlere sadece bir kez atama yapılır, sonra değiştirilemez.
  - Yapıcı metot içerisinde ilk atama yapılır.
  - Tüm iş parçacıkları tarafından aynı değerin görüleceği garanti edilir.
  - Bu durum, explicit senkronizasyon ihtiyacını ortadan kaldırır.
- volatile değişkenlerin değerleri değişebilir.
  - Ancak okuma işleminin en son yazılan değeri göreceği garanti edilir.
  - Bu durum, explicit senkronizasyon ihtiyacını ortadan kaldırır.
- Bir değişkenin hem volatile hem final tanımlanmasına **izin verilmez**.

Açık Kilitleme

- Karmaşık senkronizasyon senaryoları ve ileri düzey kontrol için kullanılır.
- ReentrantLock ve ReadWriteLock, gelişmiş ve özelleştirilebilir kilitleme mekanizmalarını sağlar.
- Esneklik ve performans iyileştirmeleri sağlar.
- Kullanırken dikkatli olunmalıdır.

Açık Kilitleme

```
public class Ornek {

   private final Lock lock = new ReentrantLock();

   public void calistir() {
      lock.lock();
      try {
         // kritik bölge
         // ...
      } finally {
         lock.unlock();
      }
   }
}
```

Kilidi finally Bloğunda Aç

- Kilidi her durumda serbest bırakabilmek için finally bloğunda kullanılmalı.

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

trylock() Metodu ile Kilidi Alma

- tryLock() metodu, kilidi almaya çalışır. Alınırsa, kritik bölgeye erişim sağlar.

```
public void calistir() {
  Lock kilit = ...;
  if (kilit.tryLock()) {
    // alındı
    try {...}
    finally {
      kilit.unlock();
    }
  } else {
    // alınamadı, başka bir şey yap
  }
}
```

Belirli Bir Süre İçinde Kilidi Alma

- tryLock(timeout, unit) metodu, belirli bir süre içinde kilidi almaya çalışır.
- Alınırsa, kritik bölgeye erişim sağlar.

```
public void calistir() {
  Lock kilit = ...;
  if (kilit.tryLock(3000, TimeUnit.MILLISECONDS)) {
    // alındı
    try {...}
    finally {kilit.unlock();}
  } else {
    // 3 saniye içinde alınamadı, başka bir şey yap
  }
}
```

ReentrantLock Sınıfı

- "Yeniden girişe izin veren" anlamına gelir, yani bir iş parçacığı kendi kilidini tekrar edebilir.
- Birden fazla ReentrantLock kullanılabilir.
- İç içe geçmiş kritik bölgelere erişim sağlanabilir.
- ReentrantLock sınıfı, adil kilitleme (fair lock) özelliği sağlar.

ReentrantLock Sınıfı

```
ReentrantLock kilit1 = new ReentrantLock();
ReentrantLock kilit2 = new ReentrantLock();
// ...
kilit1.lock();
try {
  // kritik bölgeye erişim
  kilit2.lock();
  try {
    // iç içe kritik bölgeye erişim
  } finally {
    kilit2.unlock();
  }
} finally {
  kilit1.unlock();
}
```

ReadWriteLock Sınıfı

- İş parçacıkları, veriyi okurken birbirini engellemez.
- Paralel okumaya izin verir.
- Bir iş parçacığı, veri yazarken diğerlerinin okuma yapması engellenir.
- Bu, yazma işlemlerini güvenli ve atomik hale getirir.
- Veriyi sık okuma, nadiren yazma senaryolarında performansı artırır.

ReadWriteLock Sınıfı

```
ReadWriteLock kilit = new ReentrantReadWriteLock();
// Okuma kilidi al
kilit.readLock().lock();
try {
   // Veriyi oku
} finally {
   kilit.readLock().unlock();
}

// Yazma kilidi al
kilit.writeLock().lock();
try {
   // Veriyi yaz
} finally {
   kilit.writeLock().unlock();
}
```

Açık (Explicit) ve Dolaylı (Implicit) Kilitleme

- Açık:
  - Geliştirici tarafından kontrol edilir.
  - Lock arabirimi ve ReentrantLock sınıfı kullanılır.
  - Gelişmiş kontrol ve özelleştirme sunar.
  - *Support non-blocking conditional acquisition*
- Dolaylı:
  - Otomatik olarak yönetilir.
  - synchronized anahtar kelimesi kullanılır.
  - *Are always reentrant*
  - Basit senaryolarda, kodun daha anlaşılır olması gerektiğinde kullanılır.

Durum (condition) Senkronizasyonu

- Ortak bir kaynağa
  - Sadece belirli bir koşulda
    - Güvenli bir şekilde erişmeyi sağlar.
- Kontrollü bir güncelleme işlemi için nesne kilidine ihtiyaç var.
- Ayrıca, koşul sağlanana kadar beklemeniz gerekiyor.
- Fakat, kilidi tutarken bekleyemezsiniz.
- Dolayısıyla, bu problem için Java desteğine ihtiyaç var!
  - Object.wait ve Object.notify
  - The Condition Interface
  - Synchronization Objects

Object.wait ve Object.notify

- Her nesnenin bir bekleme kümesi vardır.
- Bir nesne üzerinde kilidi tutan iş parçacığı wait() çağırabilir.
- wait() çağrısı, iş parçacığını bekleme kümesine ekler, kilidi serbest bırakır ve iş parçacığını bloke eder.
- İş parçacığı, notify() veya notifyAll() çağrılana kadar kümede bekler.
- wait çağrısı üzerindeki zaman aşımı sağlandıysa, geçerli olabilir.
- Bekleme kümesinden kaldırıldıktan sonra, iş parçacığı kaldığı yerden devam eder.

BlockingQueue

```
public class BlockingQueue {
  private Object[] veri;
  private int bas = 0;
  private int son = 0;
  private int buyukluk = 0;

  public void Buffer(int boyut) {
    veri = new Object[boyut];
  }

  public synchronized int getSize() {
    return buyukluk;
  }
```

BlockingQueue

```
public synchronized void add(Object oge) throws Exception {

  while (buyukluk == veri.length) {
    wait();
  }
  veri[son] = oge;
  son = (son + 1) % veri.length;
  buyukluk++;
  notifyAll();
}
```

BlockingQueue

```
public synchronized Object remove() throws Exception {

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

Condition Arayüzü

- Bir iş parçacığının, durum değişene kadar beklemesini, durum değiştiğinde kaldığı yerden devam etmesini sağlar.
- Condition arayüzü, wait ve notify işlevselliği sağlar.
- ReentrantLock ile beraber, durum değişikliklerini kontrol eder.
- await() metodu, bir durumu beklerken iş parçacığını askıya alır.
- signal() ve signalAll() metotları, diğer bekleyen iş parçacıklarını uyandırarak devam etmelerini sağlar.

BlockingQueue

```
public 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 void Buffer(int boyut) {
    veri = new Object[boyut];
  }
```

BlockingQueue

```
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();
  }
}
```

BlockingQueue

```
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();
  }
}
```

Senkronizasyon Nesneleri

- Çoklu iş parçacıkları arasında düzeni sağlar.
- CountDownLatch, CyclicBarrier, Exchanger, Semaphore gibi araçlar, farklı senkronizasyon senaryolarına uygun çözümler sunar.

CountDownLatch

- Belirli bir sayıda olayın gerçekleşmesini bekleyen iş parçacıklarını senkronize eder.
- counter (Sayaç): İlk başta belirlenen bir sayı.
- countDown (Azaltma): Olay gerçekleştiğinde sayaç azaltılır.
- await (Bekle): Sayaç sıfır kadar iş parçacıkları bekler.
- *A CountDownLatch is a use-once object. The count never increases.*
- *Calling countDown() when the count is zero has no effect.*

CountDownLatch

```
public static void main(String[] args) throws Exception {
  int sayac = 3;
  CountDownLatch mandal = new CountDownLatch(sayac);
  for (int i = 0; i < sayac; i++) {
    new Thread(() -> {
      // Olay gerçekleştiğinde sayaç azaltılır.
      mandal.countDown();
    }).start();
  }
  // iş parçacıklarının tamamlanması beklenir.
  mandal.await();
  System.out.println("Tüm olaylar tamamlandı!");
}
```

CyclicBarrier

- Belirli bir noktada bir araya gelmek için bir grup iş parçacığını bekletir.
- Barrier (Bariyer): İş parçacıklarının beklediği noktadır.
- Party (Grup): Bir araya gelmeyi bekleyen iş parçacıklarının sayısıdır.
- Await (Bekleme): Parti sayısına ulaşana kadar iş parçacıkları bekler.
- *Barriers can be broken for several reasons: exception, timeout, interrupt.*
- *If the barrier breaks, all threads are released by having the await().*

CyclicBarrier

```
public static void main(String[] args) {
  int partiSayisi = 3;

  CyclicBarrier bariyer = new CyclicBarrier(partiSayisi, () -> {
    System.out.println("Parti tamamlandı!");
  });

  for (int i = 0; i < partiSayisi; i++) {
    new Thread(() -> {
      try {
        // iş parçacıklarını bekle
        bariyer.await();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }).start();
  }
}
```

Exchanger

- İki iş parçacığının veri veya nesne alışverişi yapması için kullanılır.
- exchange() metodu, iki iş parçacığının veri alışverişini başlatır.
- İş parçacığı, diğerinin exchange işlemini tamamlamasını bekler.
- *An exchanger is more like an object for communication.*

Exchanger

```
Exchanger<String> exchanger = new Exchanger<>();
new Thread(() -> {
  try {
    // İlk iş parçacığı veri gönderir
    String veri = "Veri 1";
    System.out.println("veri gönderildi: " + veri);
    Thread.sleep(2000);
    String alinan = exchanger.exchange(veri);
    System.out.println("veri alındı: " + alinan);
  } catch (Exception e) {
    e.printStackTrace();
  }
}).start();
```

Exchanger

```
new Thread(() -> {
  try {
    // İkinci iş parçacığı veri gönderir
    String veri = "Veri 2";
    System.out.println("gönderildi: " + veri);
    Thread.sleep(1000);
    String alinan = exchanger.exchange(veri);
    System.out.println("veri alındı: " + alinan);
  } catch (Exception e) {
    e.printStackTrace();
  }
}).start();
```

Semaphore

- Kaynaklara eşzamanlı erişimi sınırlamak ve senkronizasyon için kullanılır.
- acquire(), izin almak için kullanılır ve izin alınana kadar bekler.
- release(), izinleri serbest bırakmak için kullanılır.
- İzinler adil bir şekilde paylaşılır.
- *tryAcquire() without a timeout can break fairness.*
- *One good use of a semaphore is to limit the number of threads.*

Semaphore

```
public static void main(String[] args) {
  int izinSayisi = 3;
  Semaphore semaphore = new Semaphore(izinSayisi);
  for (int i = 0; i < 5; i++) {
    new Thread(() -> {
      try {
        semaphore.acquire();
        // Kritik bölgeye erişim sağla
        Thread.sleep(2000);
        semaphore.release();
      } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }).start();
}
```

Atomic Sınıflar

- *atomic object is like a volatile field plus atomic compareAndSet operation.*
- Çoklu iş parçacığı ortamında değişkenlere güvenli erişimi sağlar.
- AtomicInteger, AtomicLong, AtomicBoolean gibi sınıflardır.
- Değişkenlerin değeri atomik olarak kontrol edilip güncellenir.
- İşlemler yarıda kesilemez.

```
AtomicInteger sayac = new AtomicInteger(0);
int deger = sayac.incrementAndGet();
```

Kesme (Interrupting)

- interrupt(), bir iş parçacığının normal çalışmasını durdurur.
- interrupted() ile bir iptal işlemi yapıldı mı kontrol edilir.
- InterruptedException, sleep() veya wait() sırasında iptal edildiğinde fırlatılır.
- *Thread t1 can call t2.interrupt() to interrupt t2 from blocking on something.*

```
try {
  while (!Thread.interrupted()) {
    // İş parçacığının çalışma mantığı
  }
} catch (InterruptedException e) {
  System.out.println("İş parçacığı iptal edildi!");
}
```

Suspension

- *First, don't use the suspend() method! It's deprecated.*
- İş parçacığının bir süre veya koşul altında geçici olarak durdurulmasıdır.
- İş parçacığı, askıya alındığında belirli bir süre bekleyebilir
- wait() ve notify() metotları, iş parçacığının bir koşulu beklemesini ve uyandırılmasını sağlar.

ThreadLocal Sınıfı

- Her iş parçacığı için ayrı bir kopyaya sahip olan yerel değişkenlerdir.
- Bu değişkenler, her iş parçacığı tarafından bağımsız olarak kullanılır.
- initialValue(), her iş parçacığı için ayrı bir başlangıç değeri sağlar.

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

Thread Pools

- Kullanılacak iş parçacıklarını önceden oluşturup yöneten bir yapıdır.
- *A thread pool is a fixed collection of threads that you submit tasks to.*
- İş parçacıklarının yeniden kullanımını sağlar.
- ExecutorService, iş parçacığı havuzunda çalışan işlemleri kontrol eder.
- ThreadPoolExecutor, iş parçacığı havuzunu oluşturur ve yönetir.
- *Thread creation is expensive. "Reusing" threads can help a lot.*

SON