Java'da Multithreading

Ufuk Uzun

Ön Söz

Merhaba,

13 bölüm 48 sayfadan oluşan bu doküman, Udemy'de John Purcell tarafından ücretsiz olarak yayınlanmakta olan Java Multithreading adlı kursu temel alarak, Şubat ayından bu yana yazdığım blog yazılarımın bir derlemesidir. Multithreading konusuna -tıpkı benim gibi- yeni başlayacak arkadaşlar için iyi bir temel oluşturacağını umuyorum.

Okuyan herkese faydalı olması dileğiyle,

Ufuk Uzun
İstanbul, Türkiye
Mayıs 2015
ufukuzun.ce@gmail.com

Bölüm 1: Merhaba Thread Kardeş

Thread, başka iş parçacıkları ile eş zamanlı çalışabilen bir iş parçacığı olarak ifade edilebilir. Türkçe'ye çevirecek olursak "multithreading"i "eş zamanlılık", "thread"i ise "iş parçacığı" olarak ifade edebiliriz.

Bu bölümde Java'da threadler oluşturmanın ve çalıştırmanın iki basit yolunu göreceğiz.

Birinci yol "Thread" sınıfını genişleten (extend) bir sınıf oluşturmak ve "run()" metodunu gerçekleştirerek (implement) threadin yapmasını istediğimiz işi bu "run()" metodu içerisinde kodlamaktır:

```
class Runner extends Thread {

    @Override
    public void run() {
        // Thread çalıştığında yapılması istenen işlemler buraya yazılır
    }

}
```

Diyelim ki run metodumuzun gerçekleştirimi şu şekilde olsun:

```
@Override
public void run() {
    for (int i = 0; i < 10; i++) {
        System.out.println("Merhaba " + i);
    }
}
```

Şimdi threadimizin çalışmasını test etmek için "main" metoduna sahip bir sınıf oluşturalım. Bir "Runner" örnek değişkeni oluşturup "start()" metodu ile threadi işletelim:

```
public class Application {

    public static void main(String[] args) {
        Runner runner = new Runner();
        runner.start();
    }

}
```

Çıktı şu şekilde olacaktır:

```
Merhaba 0
Merhaba 1
Merhaba 2
Merhaba 3
Merhaba 4
Merhaba 5
Merhaba 6
Merhaba 7
Merhaba 8
Merhaba 9
```

DİKKAT! Threadi çalıştırmak için "run()" metodu yerine "start()" metodunu kullanıyoruz. "run()" metodunu çağırmamız halinde "runner"ın ifade ettiği işlemler ayrı bir thread içinde çalıştırılmak yerine o an geçerli thread içerisinde yani "main" metodunu çalıştıran ana thread içerisinde çalışacaktı. "start()" metodunu çağırdığımızda ise "runner"ın ifade ettiği işlemler kendi threadi içerisinde çalıştırılır.

Buraya kadar bir sürpriz yok. "run()" içine yazdığımız o döngüyü "main" metodu içine yazsak da aynı çıktıyı alırdık. Farkı görmek için bir çoklu thread (multi thread) örneği yapalım:

```
public static void main(String[] args) {
    Runner runner1 = new Runner();
    Runner runner2 = new Runner();

    runner1.start();
    runner2.start();
}
```

"main" metodu içinde iki adet thread oluşturduk ve onları birbiri ardına başlattık. Bu sefer çıktı aşağıdakine benzer şekilde olacaktır. (Eş zamanlılıktan dolayı, her çalıştırdığınızda çıktının biraz değiştiğini görebilirsiniz.)

```
Merhaba 0
Merhaba 1
Merhaba 0
Merhaba 1
Merhaba 2
Merhaba 3
Merhaba 4
Merhaba 5
Merhaba 6
Merhaba 7
Merhaba 8
```

```
Merhaba 9
Merhaba 2
Merhaba 3
Merhaba 4
Merhaba 5
Merhaba 6
Merhaba 7
Merhaba 8
Merhaba 9
```

"run()" metodu içindeki for döngüsünü threadin adınını yazdıracak şekilde düzenleyip threadlerin birbirini beklemeden eş zamanlı çalışarak birbirlerinden bağımsız olarak nasıl çalıştıklarını görelim:

```
for (int i = 0; i < 10; i++) {
    System.out.println("Merhaba " + i + " - " + this.getName());
}
```

Çıktı aşağıdakine benzer şekilde olacaktır:

```
Merhaba 0 - Thread-0
Merhaba 1 - Thread-0
Merhaba 2 - Thread-0
Merhaba 3 - Thread-0
Merhaba 0 - Thread-1
Merhaba 1 - Thread-1
Merhaba 2 - Thread-1
Merhaba 3 - Thread-1
Merhaba 4 - Thread-1
Merhaba 5 - Thread-1
Merhaba 6 - Thread-1
Merhaba 7 - Thread-1
Merhaba 8 - Thread-1
Merhaba 4 - Thread-0
Merhaba 9 - Thread-1
Merhaba 5 - Thread-0
Merhaba 6 - Thread-0
Merhaba 7 - Thread-0
Merhaba 8 - Thread-0
Merhaba 9 - Thread-0
```

Çıktıdan da anlaşılacağı üzere Thread-1 yani "runner2" çalışmak için Thread-0'ı yani "runner1"i beklemedi.

İşte eş zamanlı programlamayı önemli kılan nokta da bu. Şöyle ki; işlemci çekirdekleri kendilerine atanan işlemleri birbirinden bağımsız olarak, eş zamanlı çalıştırdı. Böylece normalde tek çekirdek ile 2T sürede bitebilecek iş T sürede bitirildi.

Yukarıdaki "DİKKAT!" notumuza dönecek olursak, eğer "start()" yerine "run()" metodunu kullansaydık, işlemler ayrı threadler yerine o an geçerli thread içinde çalıştırılacağından iki Runner örnek değişkeninin ihtiva ettiği işler her zaman sırayla çalıştırılacaktı. Yani önce "runner1" sonra "runner2" koşturulacaktı. Yani şöyle bir "main" metodu için,

```
public static void main(String[] args) {
    Runner runner1 = new Runner();
    Runner runner2 = new Runner();

    runner1.run();
    runner2.run();
}
```

çıktı her zaman aşağıdaki gibi olacaktır:

```
Merhaba 0 - Thread-0
Merhaba 1 - Thread-0
Merhaba 2 - Thread-0
Merhaba 3 - Thread-0
Merhaba 4 - Thread-0
Merhaba 5 - Thread-0
Merhaba 6 - Thread-0
Merhaba 7 - Thread-0
Merhaba 8 - Thread-0
Merhaba 9 - Thread-0
Merhaba 0 - Thread-1
Merhaba 1 - Thread-1
Merhaba 2 - Thread-1
Merhaba 3 - Thread-1
Merhaba 4 - Thread-1
Merhaba 5 - Thread-1
Merhaba 6 - Thread-1
Merhaba 7 - Thread-1
Merhaba 8 - Thread-1
Merhaba 9 - Thread-1
```

Bir thread oluşturmanın ve başlatmanın ikinci yolu ise Thread kurucusuna "Runnable" arabirimini (interface) gerçekleştiren bir sınıf örneği geçmektir.

"Runnable" arabirimi tek bir metot imzası içerir:

```
void run();
```

"Runner" sınıfımızı "Runnable" arabirimini gerçekleştiren bir sınıf olarak yeniden yazarsak:

```
class Runner implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Merhaba " + i);
        }
    }

}
```

Bu durumda "main" metodumuz şu şekilde değişecektir:

```
public static void main(String[] args) {
    Thread thread1 = new Thread(new Runner());
    Thread thread2 = new Thread(new Runner());

    thread1.start();
    thread2.start();
}
```

Yeni threadler oluşturmak için Thread sınıfının uygun kurucusuna Runnable arabirimini gerçekleştiren bir sınıfın örneği geçtiğimiz dikkatinizi çekmiştir. Yukarıdaki "main" metodu çalıştırıldığında, threadler eş zamanlı çalışacağından, çıktımız yine aşağıdakine benzer şekilde -düzensiz- olacaktır:

```
Merhaba 0
Merhaba 1
Merhaba 2
Merhaba 0
Merhaba 1
Merhaba 2
Merhaba 3
Merhaba 4
Merhaba 5
Merhaba 6
Merhaba 7
Merhaba 8
```

```
Merhaba 9
Merhaba 3
Merhaba 4
Merhaba 5
Merhaba 6
Merhaba 7
Merhaba 8
Merhaba 9
```

Eğer "Runnable" gerçekleştirimini bir kere kullanacaksak anonim sınıf da (anonymous class) kullanabiliriz:

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Merhaba");
    }
});
```

Eğer Java 8 kullanıyorsanız, yukarıdaki kullanımı bir adım daha ileri götürerek Lambda kullanabilirsiniz. Runnable arabirimi bir @FunctionalInterface olduğu için, Runnable arabirimi bekleyen herhangi bir metoda Runnable arabiriminin yapısına uygun bir Lambda geçilebilir. Lambda kullandığımızda bakın nasıl da o 6 satırlık kod parçası tek satırlık bir kod parçasına dönüşüyor:

```
Thread thread = new Thread(() -> System.out.println("Merhaba"));
```

Bölüm 2: Threadlerin Senkronizasyonuna Giriş

Aynı veri ile ilgilenen birden fazla threadin veriye erişiminin düzenlenmesine threadlerin senkronizasyonu denir. Threadlerin senkronizasyonu sırasında karşılaştığımız zorlukların ilki bazı durumlarda verinin performans gerekçesiyle otomatik olarak önbelleklenmesidir (caching). Bir örnek ile üstesinden gelmemiz gereken bu durumu açıklayalım. Thread sınıfını genişleterek oluşturduğumuz Processor sınıfımız aşağıdaki gibi olsun:

```
class Processor extends Thread {

    private boolean running = true;

    @Override
    public void run() {
        while (running) {
            System.out.println("Merhaba");
        }
    }

}
```

"run()" metodunun içeriğini incelersek bu sınıftan bir örnek oluşturup "start()" metodunu çağırdığımızda başlatılacak olan threadin sonsuza kadar "Merhaba" yazdıracağını görebiliriz. Çünkü "running" değişkeni her zaman "true" değerindedir. Gelin yeni bir metot ekleyelim ve çağrıldığında "running" değişkeninin değerini "false" yaparak "while" döngüsünü sonlandırmış olalım, dolayısıyla thread de sonlanmış olsun:

```
class Processor extends Thread {

    ...

    public void shutdown() {
        this.running = false;
    }

}
```

"main" metodumuz ise Processor tipinde bir threadi başlatsın ve bu thread başladıktan sonra kendi threadini 10 saniye beklettikten sonra "shutdown()" metodunu çağırarak başlattığı Processor tipindeki threadi sonlandırmak istesin:

```
public static void main(String[] args) throws InterruptedException {
        Processor processor = new Processor();
```

```
        processor.start();

        Thread.sleep(10000);

        processor.shutdown();
}
```

Buradaki tehlike şu: "processor" threadi bazı sistemlerde, bazı Java gerçekleştirimlerinde "running" değişkeninin dışarıdan herhangi bir müdahale ile değiştirilmeyeceği varsayımından yola çıkarak onun değerini otomatik olarak önbelleğe atar (caching) ve her zaman değeri "true"ymuşcasına davranılır. Bu durum için İngilizce "caching thread-locally" gibi tabirler kullanılmaktadır. Eğer bu önbelleğe atma durumu bizim programımız için tutarsızlıklara sebep olacaksa (örneğin yukarıdaki gibi threadin sonlanmaması durumu) çare söz konusu veri için (yukarıdaki durumda bu veri "running" değişkenidir) "volatile" anahtar kelimesini kullanmaktadır. "volatile" kelimesi "değişken, uçucu" anlamına gelmektedir. Bir diğer anlamı da "tutarlı"dır. Tutarlıdır, yani her başvurduğumuzda değişkenin o anki gerçek değerini döner, önbellekteki değerini değil:

```
class Processor extends Thread {

    private volatile boolean running = true;

    ...

}
```

Böylece kodumuzun tüm sistemlerde, tüm Java gerçekleştirimlerinde çalışacağını garanti etmiş olduk.

Bölüm 3: "synchronized" Anahtar Kelimesi

İkinci bölümde threadlerin senkronizasyonu sırasında karşılaştığımız zorlukların birincisinden bahsetmiştik. Bu bölümde bu zorlukların ikincisini ve onu aşmak için Java'nın bize sağladığı "synchronized" anahtar kelimesini kullanmayı örnekleyeceğiz. Önce zorluğun/problemin ne olduğunu yine bir örnek ile açıklamaya çalışalım:

```
public class Application {

    private int count = 0;

    public static void main(String[] args) {
        Application application = new Application();
        application.doCount();
    }

    private void doCount() {
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    count++;
                }
            }
        });

        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    count++;
                }
            }
        });

        thread1.start();
        thread2.start();

        System.out.println("Sayaç: " + count);
    }

}
```

"Application" sınıfının "count" (sayaç) adlı, başlangıç değeri 0 olan tam sayı tipinde bir örnek değişkeni vardır. "main" metodu içinde bir "Application" nesnesi oluşturarak "doCount()"

çağrılır. "doCount()" metodu içinde, "run()" metodu içerisinde "count" değişkenini 10000 kere arttıracak birer döngü içeren iki thread oluşturulur ve başlatılır. Metodun sonunda ise "count" değişkeninin son değeri yazdırılır. Buraya kadar her şey açık. Ancak bu programı çalıştırdığımızda bizi bir sürpriz beklemektedir. Program sonlandığında konsolda 20000 görmeyi bekleyenlerimiz olacaktır. Evet bazen 20000 görebiliriz de. Ancak 511 gibi, 10987 gibi saçma sayılar gördüğünüzde siz de benim gibi şaşırmışsınızdır. Sebebi şudur: "thread1" ve "thread2" ayrı threadler oldukları için "start()" metotlarıyla başladıktan sonra ana thread, yani bu iki threadi başlatan o anki geçerli thread işleyişine devam eder ve "count" değişkeninin anlık değerini konsola yazdırır. Bu sırada "thread1" ve "thread2" sonlanmamış olabilecekleri için sayacın son değeri bazı durumlar için 20000 olamayabilir. Bu durumu çözmek için bu iki threadin "join()" metodunu çağırarak threadlere, çalışmaları bitene kadar "geçerli threadi" bekletmelerini söyleyebiliriz:

```
private void doCount() {
    ...

    thread1.start();
    thread2.start();

    try {
        thread1.join();
        thread2.join();
    } catch (InterruptedException e) {
    }

    System.out.println("Sayaç: " + count);
}
```

Programı çalıştırdığımızda ikinci sürpriz karşımıza çıkar. Programı her çalıştırdığımızda 20000 görmeyi beklerken, yine bazen 20000 görmekle birlikte, 14180 gibi, 12460 gibi, 19422 gibi sonuçlar da görürüz. Fakat bu sefer 511 gibi küçük rakamlar göremezsiniz. "join()" çağrısı ile geçerli threadi bekletmek o kadar da yavan bir çözüm değildir. Ancak sorunlarımızdan yalnızca birini çözmektedir. Sorunu anlamak için biraz daha yukarılara bakmamız gerekiyor; "run()" metodunun içine… Sorun threadlerin eş zamanlı çalışmasından kaynaklanmakta. Ama nasıl olur, amacımız threadleri eş zamanlı çalıştırmak ve böylece performans artışı sağlamak değil miydi zaten. Threadlerin bu güçlü yanı neden şimdi karşımıza "sorun" olarak çıktı? Çünkü "kontrolsüz güç güç değildir."

"run()" metoduna, "for" döngüsünün gövdesine baktığımızda şu satırı görmekteyiz:

```
count++;
```

Bildiğiniz üzere bu kod parçası bir söz diziminden ibarettir. Asıl karşılığı ise şu şekildedir:

```
count = count + 1;
```

Burada yapılan işlem iki adımlıdır: 1) "count" değerini oku ve bir ekle, 2) sonucu "count" değişkenine ata. Bu işlem çok çok hızlı gerçekleşmektedir. Yine de eş zamanlı çalışan threadler söz konusu olduğunda bu hız yetersiz kalabilir. Örneğin:

1. "thread1" "count"un değerini 100 olarak okur ve ona 1 ekler. Yani 101 sonucuna ulaşır.
2. "thread2" de aynı anda "count"un değerini 100 olarak okur ve ona 1 ekler. Yani o da 101 sonucuna ulaşır.
3. "thread1" bulduğu sonucu, yani 101'i "count" değişkenine yazar.
4. "thread2" bulduğu sonucu, yani 101'i "count" değişkenine yazar.
5. Sonuçta değeri 2 artması gereken "count" değişkeninin son değeri "101" olmuş olur, yani yalnızca 1 artmış olur.

Bu gibi nedenlerle "count" değişkeninin değerini arttırma işlemi aynı zamanda yalnızca bir threadin yapabileceği şekilde gerçekleştirilmelidir... Java'nın imdadımıza koşan anahtar kelimesi bu defa "synchronized" oluyor. Birinci adım olarak "count" değişkeninin değerini arttırma işlemini "Application" sınıfının bir metodu olarak ifade etmekle başlayalım ve anonim "Runnable" gerçekleştirimlerimiz içindeki "count++" satırlarımızı bu metodu çağıracak şekilde düzenleyelim:

```
public class Application {

    private int count = 0;

    ...

    public void increment() {
        count++;
    }

    private void doCount() {
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                for (int i = 0; i < 10000; i++) {
                    increment();
                }
            }
        });

        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
```

```
                for (int i = 0; i < 10000; i++) {
                    increment();
                }
            }
        });

        ...
    }

}
```

Sonra da bunu bir adım daha ileri götürerek, "synchronized" anahtar kelimesi yardımıyla, "increment()" metodunun aynı anda yalnızca bir thread tarafından çağrılabileceğini garanti edelim:

```
public synchronized void increment() {
    count++;
}
```

Böylece "count" değişkeninin değerini arttırma işlemini aynı anda yalnızca bir threadin gerçekleştirebilmesi kısıtıyla gerçekleştirmiş olduk. Bir thread "increment()" metodunu çağırdığında, o thread için işlem bitene kadar "increment()" metodunu çağırmak isteyen diğer tüm threadler bekler. Buna İngilizce terim olarak "intrinsic lock" yani yerleşik/içsel kilit mekanizması denmektedir. Artık programı her çalıştırdığımızda istisnasız olarak 20000 değerini görürüz.

Ayrıca, "count" değişkeni üzerinde işlem yapan metot "synchronized" olduğu için "count" değişkenini önceki bölümde gördüğümüz gibi "volatile" anahtar kelimesiyle belirtmedik. Bunun nedeni "synchronized" anahtar kelimesinin "count" değişkeni için "volatile" anahtar kelimesinin işlevselliğini garanti etmesidir.

Bir dip not olarak; burada gerçekleştirdiğimiz eşzamanlı threadler ile sayma gibi bir işleve ihtiyaç duyduğumuzda Java'nın dahili kütüphanelerinin içerisinde gelen AtomicInteger'ı kullanabiliriz. AtomicInteger yukarıda gerçekleştiriğimiz "thread-safe" sayma işlemini gerçekleştiren kullanıma hazır bir yardımcı sınıftır.

Bölüm 4: "synchronized" Kod Blokları

Önceki bölümde bir metodu aynı anda birden fazla threadin işletememesini istediğimizde "synchronized" anahtar kelimesini nasıl kullanabileceğimizi örneklemiştik. Bu bölümde ise "synchronized" anahtar kelimesinin imdadımıza yetiştiği bir başka durum üzerinde duracağız. Yine bir örnek ile açılışı yaparak derman arayacağımız derdi ortaya koyalım. Özet olarak iki tam sayı (integer) listemiz var ve bu listeleri rastgele sayılarla doldurmak istiyoruz:

```
public class Application {

    private List<Integer> list1 = new ArrayList<>();

    private List<Integer> list2 = new ArrayList<>();

    public static void main(String[] args) {
        Application app = new Application();

        long startTime = System.currentTimeMillis();

        app.work();

        long endTime = System.currentTimeMillis();

        System.out.println("Geçen zaman: " + (endTime - startTime));
        System.out.println("List 1'in boyutu: " + app.list1.size());
        System.out.println("List 2'nin boyutu: " + app.list1.size());
    }

    private void work() {
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                process();
            }
        });

        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
                process();
            }
        });

        thread1.start();
        thread2.start();
```

```
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
        }
    }

    private void process() {
        for (int i = 0; i < 1000; i++) {
            addNewIntegerToList1();
            addNewIntegerToList2();
        }
    }

    private void addNewIntegerToList1() {
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
        }

        list1.add(new Random().nextInt());
    }

    private void addNewIntegerToList2() {
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
        }

        list2.add(new Random().nextInt());
    }

}
```

Örnek program okumak için biraz uzun evet. Ancak yaptığı şey oldukça basit; iki thread ile "list1" ve "list2"ye 1000'er sayı eklemek. Listelere toplamda 2000'er sayı eklenmesi bekleniyor. Ayrıca "main" metodu içerisinde Application.work() metodu çağrılmadan önceki ve sonraki zamanı elde ederek yaklaşık çalışma süresini çıktı olarak yazdırıyor. Bu zamanın bize anlamlı görünmesi için daha önce gördüğümüz gibi Thread.sleep(1) çağrılarıyla ilgili threadi bekletiyor. Thread.start() ve Thread.join() çağrılarını ise zaten daha önce görmüş ve irdelemiştik. Gelelim programımızı çalıştırdığımızda -eğer şanslıysak- göreceğimiz çıktıya:

```
Geçen zaman: 2570
List 1'in boyutu: 1980
List 2'nin boyutu: 1980
```

Ancak her zaman göreceğimiz çıktı böyle olmayacaktır. Çoğu zaman aşağıdaki gibi bir çıktı görürüz:

```
Exception in thread "Thread-0" java.lang.ArrayIndexOutOfBoundsException: 15
       at java.util.ArrayList.add(ArrayList.java:459)
       at multithreading.chapter4.Application.addNewIntegerToList2(Application.java:74)
       at multithreading.chapter4.Application.process(Application.java:55)
       at multithreading.chapter4.Application.access$000(Application.java:7)
       at multithreading.chapter4.Application$1.run(Application.java:31)
       at java.lang.Thread.run(Thread.java:745)
Geçen zaman: 2556
List 1'in boyutu: 1009
List 2'nin boyutu 1009
```

"list1" ve "list2" değişkenleri -bir önceki bölümde gördüğümüz "count" değişkeni gibi- iki thread için ortak veridir (shared data). Yani iki thread aynı anda erişip üzerinde aynı anda işlem yapmaya kalkışabiliyor. Bu durum ise yukarıdakiler gibi tutarsızlıklara sebep oluyor. Bu gibi sıkıntıları çözmek için aklımıza gelen ilk yöntem -bir önceki bölümde öğrendiğimiz gibi- "addNewIntegerToList1" ve "addNewIntegerToList2" metotlarını "synchronized" yapmak olacaktır. Ve evet böyle yapmak önceki çıktılarda gördüğümüz gariplikleri ortadan kaldıracaktır:

```
...

private synchronized void addNewIntegerToList1() {
    try {
        Thread.sleep(1);
    } catch (InterruptedException e) {
    }

    list1.add(new Random().nextInt());
}

private synchronized void addNewIntegerToList2() {
    try {
        Thread.sleep(1);
    } catch (InterruptedException e) {
    }

    list2.add(new Random().nextInt());
}
```

Programımızı bu haliyle çalıştırdığımızda beklendiği gibi listelerin boyutunu her seferinde 2000 olarak görürüz:

```
Geçen zaman: 4960
List 1'in boyutu: 2000
List 2'nin boyutu: 2000
```

Ancak bu defa da çalışma süresi iki katına çıktı. Bunun nedeni daha önce sözünü ettiğimiz içsel kilittir (intrinsic lock). Örnek üzerinden açıklayacak olursak; evet "app" nesnesinin "synchronized" yaptığımız "addNewIntegerToList1" metodu bir thread tarafından çağrıldığında başka bir threadin "addNewIntegerToList1" metodunu çağırması için ilk çağıran threadin yaptığı çağrının sonlanmasını beklemesi gerekir. Ve aynı şekilde "app" nesnesinin "synchronized" yaptığımız "addNewIntegerToList2" metodu bir thread tarafından çağrıldığında başka bir threadin "addNewIntegerToList2" metodunu çağırması için ilk çağıran threadin yaptığı çağrının sonlanmasını beklemesi gerekir. Buraya kadar gayet normal. Fakat burada beklenmedik bir şey söz konusudur. Metot tanımında belirtilen "synchronized" kelimesi o metoda yapılan çağrıları bekletmek için içsel kilidi kullanır. Ve içsel kilit bir nesne için yalnızca bir tanedir. Bu şuna sebep olur: Bir thread "addNewIntegerToList1" çağrısı yaptığında "addNewIntegerToList1" metoduna yapılacak çağrıları bekletirken -içsel kilidin bir nesne için sadece bir tane olması sebebiyle- "addNewIntegerToList2" metoduna yapılacak çağrıları da bekletmiş olur. Fakat "list1" ve "list2" birbirinden ayrı ortak veriler oldukları için, birine erişildiğinde diğerine erişimin kısıtlanması gereksizdir. Bu sefer imdadımıza yetişen çoklu kilitler (multiple locks) ve "synchronized" kod bloklarıdır. "synchronized" kod blokları farklı kod parçaları için farklı kilitler belirtebilmemize olanak sağlar. İlk iş iki adet kilit nesnesi oluştururuz, ve "list1" ve "list2" listelerine yeni eleman eklediğimiz yerleri bu birbirinden ayrı kilitleri kullanacak şekilde birer "synchronized" kod bloğu olarak ifade ederiz:

```
public class Application {

    private Object lock1 = new Object();

    private Object lock2 = new Object();

    ...

    private void addNewIntegerToList1() {
        synchronized (lock1) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
            }

            list1.add(new Random().nextInt());
        }
    }
```

```
    private void addNewIntegerToList2() {
        synchronized (lock2) {
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
            }

            list2.add(new Random().nextInt());
        }
    }

}
```

Programımızı bu haliyle çalıştırdığımızda çalışma süresinin yaklaşık yarı yarıya düştüğünü görebiliriz:

```
Geçen zaman: 2787
List 1'in boyutu: 2000
List 2'nin boyutu: 2000
```

İçsel kilitteki gibi tek kilit olduğu durumu bu yeni yapı ile gözlemlemek amacıyla iki kod bloğunda da örneğin "lock1" kilidini belirtelim:

```
...

private void addNewIntegerToList1() {
    synchronized (lock1) {
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
        }

        list1.add(new Random().nextInt());
    }
}

private void addNewIntegerToList2() {
    synchronized (lock1) {
        try {
            Thread.sleep(1);
        } catch (InterruptedException e) {
        }

        list2.add(new Random().nextInt());
```

```
    }
}
```

Bu durumda, tıpkı içsel kilitte olduğu gibi, çalışma süresi tekrar yaklaşık iki katına çıkacaktır:

```
Geçen zaman: 5049
List 1'in boyutu: 2000
List 2'nin boyutu: 2000
```

Bölüm 5: Thread Havuzları (Thread Pools)

Diyelim ki "x" tane görevimiz olsun ve mesela bu görevleri "y"li threadler halinde yerine getirmek isteyelim. Java'da bunu nasıl gerçekleştirirdik? Java'nın bu ihtiyaca karşılığı thread havuzlarıdır, yani İngilizce tabiriyle "Thread Pools".

Örneğin aşağıdaki gibi bir Processor sınıfımız olsun. "run()" metodu daha önce öğrendiğimiz üzere kendisini işletecek threadin yerine getireceği görevi ifade ediyor:

```
class Processor implements Runnable {

    private int id;

    public Processor(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        System.out.println("Başlıyor: " + id);

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
        }

        System.out.println("Tamamlandı: " + id);
    }

}
```

Processor sınıfının "run()" metodu basitçe bize başladığını bildiriyor, işini yapıyor (görüldüğü üzere işi yalnızca 5 saniye beklemek), ve nihayet bittiğini bildiriyor.

Diyelim ki Processor sınıfının ifade ettiği görevi 5 kere yapmamız gerekti. Ve donanımsal olarak 2 işlemci çekirdeğimiz mevcut, yani aynı anda 2 thread işletebiliriz. Elle 2 thread oluşturup ve bu threadlerin sonlanmasını bekleyip 2 thread daha oluşturarak devam edebiliriz. Ancak böyle yapmak yerine Java'nın bize sağladığı yardımcı sınıflar (util classes) yardımıyla 2 thread büyüklüğünde bir thread havuzu oluşturup yapmak istediğimizi çok daha kolay ve olması gerektiği şekilde yapabiliriz.

2 boyutlu bir thread havuzu oluşturmak için java.util.concurrent.Executors sınıfının "newFixedThreadPool()" adlı statik metodundan yararlanacağız:

```
public class Application {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(2);

        for (int i = 1; i <= 5; i++) {
            executorService.submit(new Processor(i));
        }

        executorService.shutdown();

        System.out.println("Tüm görevler eklendi.");

        try {
            executorService.awaitTermination(1, TimeUnit.DAYS);
        } catch (InterruptedException e) {
        }

        System.out.println("Tüm görevler tamamlandı.");
    }

}
```

"Executors.newFixedThreadPool(2)" çağrısı bize 2 threadlik bir havuz oluşturup görevlerimizi kabul edip işletecek olan bir java.util.concurrent.ExecutorService nesnesi döner. "ExecutorService.submit()" metodu ile Processor tipindeki görevlerimizi görev işleticiye (executorService) ekleriz. "ExecutorService" havuzunda yer olduğu sürece kendisine eklenen Runnable nesneleri ile yeni birer thread oluşturur ve onu başlatır. Örneğimizdeki gibi bir "ExecutorService" şöyle çalışmaktadır: Kendisine eklenen ilk görevi hemen işletmeye başlar. İkinci görev eklenir eklenmez hemen onu da işletmeye başlar. Ancak üçüncü görev geldiğinde önce havuzunda yer açılmasını bekler, yani ilk iki görevden birinin bitmesini. Bir diğer deyişle havuzdaki görevlerden biri tamamlanana kadar diğer görevler kuyrukta bekler. Ve tüm görevler bitene kadar bu şekilde devam eder.

"ExecutorService.shutdown()" çağrısı önemlidir. Şunu ifade eder: Görev işleticiye (executorService) yeni görev kabul etme ve mevcuttaki tüm görevler bittiğinde görev işleticiyi sonlandır. Eğer bu çağrı yapılmazsa, yukarıdaki program sonlanmayacak, yeni görevler beklemeye devam edecektir. Biz de örneğimizde "shutdown()" çağrısı sonrası bunu belirtmek amacıyla konsola "Tüm görevler eklendi." şeklinde bir bilgilendirme mesajı yazdırıyoruz. "shutdown()" çağrısı sonrasında "executorService.submit()" çağrısı ile yeni görev eklemeye kalkarsanız "java.util.concurrent.RejectedExecutionException" tipinde bir istisna (exception) fırlatıldığını görebilirsiniz.

Peki tüm görevlerin tamamlanmasının ardından bir şeyler yapmak istersek? Bunun için "ExecutorService.awaitTermination()" metodu kullanılır. "awaitTermination()" metodu ile bir zaman sınırlaması belirtilir. Metodun çağrısında belirtilen zaman her koşulda beklenecek zamanı değil maksimum beklenecek zamanı belirtir. Örneğimizde bu zamanı 1 gün olarak belirttik. Bu şu demek oluyor: Tüm görevler tamamlanana kadar bekle. Eğer 1 günde tüm görevler tamamlanmazsa görev işleticiyi sonlandır.

Şimdi programı çalıştırıp çıktıyı görme zamanı:

```
Başlıyor: 1
Başlıyor: 2
Tüm görevler eklendi.
Tamamlandı: 2
Tamamlandı: 1
Başlıyor: 3
Başlıyor: 4
Tamamlandı: 3
Tamamlandı: 4
Başlıyor: 5
Tamamlandı: 5
Tüm görevler tamamlandı.
```

"run()" metodunda 5'er saniye beklettiğimiz için thread havuzlarının çalışma mantığını görsel olarak da gözlemleyebiliriz.

1. ve 2. görevin, 3. ve 4. görevin tamamlanma sırası kendi içlerinde değişebileceği için çıktı da değişebilir. Ancak threadlerin çalışmaya başlama sıraları her zaman aynı olacaktır.

Bölüm 6: Geri Sayım İçin CountDownLatch Yardımcı Sınıfı

Java'nın dahili kütüphanelerinde "multithreading" programlamaya dair pek çok yardımcı sınıf mevcuttur. (java.util.concurrent.*) Bunlardan biri de bu bölümde ele alacağımız "CountDownLatch" sınıfıdır. "Count down" "geri sayım", "latch" "kapı sürgüsü/kilit mandalı" manalarına gelmektedir.

Geri sayım tamamlanıncaya kadar CountDownLatch örneğinin "await()" metodunu çağıran tüm threadler bekletilir. Geri sayım tamamlandığında bir anlamda kapı sürgüsü açılır ve bekleyen threadler işlemeye devam eder. Örneğin "n" sayıda threadin işlerini bitirdiklerini bildirene kadar (yani her biri sayacı bir azaltana ve nihayetinde sayacın değeri sıfıra ulaşana kadar) ana threadin bekletilmesi istediğimiz bir durumda kullanılabilir:

```
public class Application {

    public static void main(String[] args) {
        CountDownLatch latch = new CountDownLatch(3);

        ExecutorService executorService = Executors.newFixedThreadPool(3);

        for (int i = 0; i < 3; i++) {
            executorService.submit(new Thread(new Processor(latch)));
        }

        try {
            latch.await();
        } catch (InterruptedException e) {
        }

        System.out.println("Program bitti.");
    }

}

class Processor implements Runnable {

    private CountDownLatch latch;

    public Processor(CountDownLatch latch) {
        this.latch = latch;
    }

    @Override
```

```
    public void run() {
        System.out.println("Thread başladı.");

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
        }

        latch.countDown();
    }

}
```

Örneğimiz oldukça basit. 3'ten (n) geriye sayacağımız için CountDownLatch kurucusuna 3'ü geçtik. Önceki bölümde gördüğümüze benzer şekilde 3 threadlik bir havuz oluşturup 3 adet thread ekledik. Threadlerimize Runnable örneği olarak birer Processor örneği geçtik. Threadlerin her biri işini bitirdikten sonra (yani 5 saniye bekledikten sonra) CountDownLatch örneğinin "countDown()" metodunu çağırarak sayacı bir azaltacaklar. CountDownLatch halihazırda "thread-safe" olduğu için, önceki bölümlerde gördüğümüz gibi senkronizasyon için ayrıca bir çaba sarf etmemiz gerekmez.

Örneğimizde "CountDownLatch" örneğinin "await()" metodu yalnızca ana thread tarafından çağrılıyor. Ana thread her bir thread işini bitirip sayacı bir azaltana, yani sayaç sıfıra ulaşıncaya kadar bekletilir, sayaç sıfırlandığındaysa çalışmasına devam eder, ve "Program bitti." yazdırılır. Programı çalıştırdığınızda, Thread.sleep(5000) çağrısı sayesinde bu durumu siz de rahatlıkla gözlemleyebilirsiniz. Programın çıktısı aşağıdaki şekilde olacaktır:

```
Thread başladı.
Thread başladı.
Thread başladı.
Program bitti.
```

Son olarak belirtilmesi gereken önemli bir nokta; CountDownLatch yeniden kullanılabilir (reusable) değildir. Bir diğer deyişle sayacı bir kere sıfıra ulaştığında bir daha kullanılamaz. Yeniden bir sayma işlemine ihtiyaç duyduğumuzda yeni bir CountDownLatch örneği oluşturmamız gerekir.

Bölüm 7: Producer-Consumer Yapısı

Producer (Üretici) ve Consumer (Tüketici) yapısı hem günlük hayatta hem de programlama yaparken sıkça karşılaşabileceğimiz bir yapıdır. Producer kuyruğa bir şeyler ekler, Consumer ise bu kuyrukta bir şeyler oldukça sırayla alır ve ne yapması gerekiyorsa yapar. Günlük hayattan bir örnek verecek olursak; bir bankada müşteriler için sıra numarası üreten cihaza Producer, işlem görmemiş sıra numaralı müşteri oldukça onları LED sıra göstergeleri aracılığıyla çağırıp işlem yapan vezne görevlisi de Consumer olarak düşünülebilir.

Siz de fark etmişsinizdir, Producer ve Consumer birbirinden bağımsız iki ayrı thread halinde gerçekleştirilebilir. Ancak burada yine ortak bir veri kaynağımız olduğunu gözden kaçırmayalım: Kuyruk (Queue). Yine yardımımıza yetişen java.util.concurrent.* paketinden "thread-safe" bir yardımcı sınıf olan "BlockingQueue" oluyor. BlockingQueue aslında bir arabirim (interface) ve biz onun gerçekleştirimlerinden (implementation) biri olan "ArrayBlockingQueue" sınıfını kullanacağız. ArrayBlockingQueue tipinde bir kuyruk oluştururken bir kapasite belirtiriz. Bu kapasite dolu ise kuyruğa yeni eleman eklemek isteyen thread bekletilir. Benzer şekilde eğer kuyrukta eleman yoksa kuyruktan eleman almak isteyen thread kuyrukta eleman oluncaya kadar bekletilir.

Şimdi Producer-Consumer yapısını ve ArrayBlockingQueue sınıfının kullanımını örnekleyeceğimiz programımızı inceleyelim:

```
public class Application {

    private static BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(10);

    public static void main(String[] args) throws InterruptedException {
        Thread producerThread = new Thread(() -> {
            produce();
        });

        Thread consumerThread = new Thread(() -> {
            consume();
        });

        producerThread.start();
        consumerThread.start();

        producerThread.join();
        consumerThread.join();
    }

    private static void produce() {
        Random random = new Random();
```

```
        while (true) {
            try {
                queue.put(random.nextInt(100));
            } catch (InterruptedException e) {
            }
        }
    }

    private static void consume() {
        while (true) {
            try {
                Thread.sleep(100);
                Integer value = queue.take();
                System.out.print("Alınan sayı: " + value
                                 + ", Kuyruğun boyutu: " + queue.size());
            } catch (InterruptedException e) {
            }
        }
    }

}
```

Öncelikle 10 eleman kapasiteli ve elemanları tam sayı olacak olan bir kuyruk oluşturuyoruz. Ardından main metodu içerisinde biri "produce()" diğeri "consume()" metodunu çağıracak iki adet thread oluşturup başlatıyoruz. "procude()" metodu bir sonsuz döngü içerisinde rastgele sayılar oluşturarak kuyruğa ekliyor. "consume()" metodu ise yine bir sonsuz döngü içerisinde (rahat gözlemleyebilmemiz için 100 mili saniye bekleyerek) kuyruktan bir sayı alıyor ve bu sayı ile kuyruğun o anki boyutunu yazdırıyor. Programı çalıştırdığımızda çıktı aşağıdakine benzer şekilde olacaktır:

```
Alınan sayı: 44, Kuyruğun boyutu: 9
Alınan sayı: 34, Kuyruğun boyutu: 10
Alınan sayı: 51, Kuyruğun boyutu: 10
Alınan sayı: 49, Kuyruğun boyutu: 10
Alınan sayı: 18, Kuyruğun boyutu: 9
Alınan sayı: 37, Kuyruğun boyutu: 9
Alınan sayı: 88, Kuyruğun boyutu: 10
Alınan sayı: 94, Kuyruğun boyutu: 9
Alınan sayı: 99, Kuyruğun boyutu: 9
Alınan sayı: 95, Kuyruğun boyutu: 9
Alınan sayı: 69, Kuyruğun boyutu: 10
Alınan sayı: 72, Kuyruğun boyutu: 9
Alınan sayı: 96, Kuyruğun boyutu: 9
Alınan sayı: 72, Kuyruğun boyutu: 10
Alınan sayı: 56, Kuyruğun boyutu: 10
```

```
Alınan sayı: 92, Kuyruğun boyutu: 10
Alınan sayı: 89, Kuyruğun boyutu: 9
...
```

Consumer 100 mili saniyede bir kuyruktan eleman aldığı için o 100 mili saniye geçene kadar Producer kuyruğu çoktan doldurmuş oluyor. Consumer kuyruktan bir eleman aldıkça eş zamanlı çalışan Producer bazen Consumer kuyruğun boyutunu henüz yazdırmamışken yerine yenisini eklemiş olabiliyor. Bu nedenle kuyruğun boyutu genelde 9 veya 10 olarak yazdırılmakta.

Programın çalışmasını rahatça gözlemleyebilmemiz için koyduğumuz "Thread.sleep(100)" çağrısını kaldırırsak kuyruk boyutunun zaman zaman farklılaştığını, hatta bazen sıfırlandığını görebiliriz:

```
...
Alınan sayı: 71, Kuyruğun boyutu: 9
Alınan sayı: 81, Kuyruğun boyutu: 8
Alınan sayı: 87, Kuyruğun boyutu: 7
Alınan sayı: 69, Kuyruğun boyutu: 6
Alınan sayı: 5, Kuyruğun boyutu: 5
Alınan sayı: 15, Kuyruğun boyutu: 4
Alınan sayı: 94, Kuyruğun boyutu: 3
Alınan sayı: 6, Kuyruğun boyutu: 2
Alınan sayı: 0, Kuyruğun boyutu: 1
Alınan sayı: 95, Kuyruğun boyutu: 0
Alınan sayı: 14, Kuyruğun boyutu: 9
Alınan sayı: 4, Kuyruğun boyutu: 9
Alınan sayı: 9, Kuyruğun boyutu: 9
...
```

Bölüm 8: Beklet ve Bildir (Wait and Notify)

Bir önceki bölümde Producer-Consumer yapısını ve bir kuyruk ile nasıl gerçekleştirebileceğimizi incelemiştik. Kuyruğa eleman eklerken veya kuyruktan eleman alırken bazı koşullara göre ilgili threadlerin bekletildiğinden söz etmiştik. Bu bölümde bu işlevselliği kendi çok threadli programlarımız için nasıl gerçekleştirebileceğimizi öğreneceğiz.

Java'da her sınıf otomatik olarak "Object" sınıfından türer. "Object" üst sınıfına ait "wait()" ve "notify()" metotları yukarıda bahsettiğimiz bekleme ve tekrar sürdürme işlevselliğini sağlayabilmek için kullanılırlar. Bu bölümde "wait()" ve "notify()" metotlarının ne işe yaradıklarına ve nasıl kullanıldıklarına basit bir örnekle giriş yapmış olacağız.

Üretici ve tüketici metotlara yani "produce()" ve "consume()" metotlarına sahip Processor sınıfımız aşağıdaki gibi olsun:

```
public class Processor {

    public void produce() throws InterruptedException {
        synchronized (this) {
            System.out.println("Üretici thread çalışıyor...");
            wait();
            System.out.println("Üretici devam ediyor...");
        }
    }

    public void consume() throws InterruptedException {
        Thread.sleep(2000);

        Scanner scanner = new Scanner(System.in);

        synchronized (this) {
            System.out.println("Tüketici thread çalışıyor...");
            System.out.print("Devam etmek için 'Enter'a basınız: ");
            scanner.nextLine();
            System.out.println("Tüketici devam ediyor...");
            notify();
            Thread.sleep(5000);
            System.out.println("Tüketici 5 saniye daha devam etti.");
        }
    }

}
```

"main" metodumuz ise bir önceki bölümdekine benzer şeyi gerçekleştirecek; biri "produce()" diğeri "consume()" metodunu işletecek iki thread oluşturup başlatmak:

```
public class Application {

    public static void main(String[] args) throws InterruptedException {
        final Processor processor = new Processor();

        Thread producerThread = new Thread(() -> {
            try {
                processor.produce();
            } catch (InterruptedException e) {
            }
        });

        Thread consumerThread = new Thread(() -> {
            try {
                processor.consume();
            } catch (InterruptedException e) {
            }
        });

        producerThread.start();
        consumerThread.start();

        producerThread.join();
        consumerThread.join();
    }

}
```

Örnek programımızın tamamı karşımızda olduğuna göre Processor sınıfımızı irdelemeye başlayalım: "produce()" metodu geçerli nesneyi kilit nesnesi (lock object) olarak kullanarak bir "synchronized" blok oluşturuyor. (Bir anlamda içsel kilit mekanizmasını kullanıyor.) Blok içerisinde, çalışmaya başladığını bildiren bir çıktı yazdırdıktan sonra o nesneye ait "wait()" metodunu çağırıyor. Bu çağrı ile geçerli threadin çalışması -sonradan devam etmek üzere- beklemeye alınıyor ve blok için belirtilen kilit serbest kalıyor. Kilit serbest kaldığı için başka threadler aynı kilide sahip bloklara girebilecekler.

İlk önce "produce()" metodundaki "synchronized" bloğa girildiğinden emin olmak için "consume()" metodunun başında "Thread.sleep(2000)" şeklinde bir çağrı yaptık. (Bunu yalnızca örneği anlaşılır kılmak için yaptık elbette.) Bu geçen sürede "produce()" metodu içerisindeki blokta "wait()" çağrısı yapıldı ve ilgili thread beklemeye alınarak blok için belirtilen kilit serbest bırakıldı. Kilidin serbest bırakılmasıyla, "produce()" metodu içerisindeki blok sonlanmamasına rağmen "consume()" metodu içerisindeki aynı kilit nesnesine sahip bloğa girilebildi. Bu bloğa girildiğinde önce bloğa girildiğine dair bir çıktı yazdırılıyor. Ardından kullanıcıdan 'Enter'a basması beklenir. Kullanıcı 'Enter'a bastıktan sonra en nihayetinde, bu

bloğa ait kilide sahip ve beklemekte olan diğer threadi -artık devam edebileceğine dair- bilgilendirmek amacıyla, "notify()" çağrısı yapılır. Programı çalıştırdığımız ve bizden istendiğinde 'Enter'a bastığımız taktirde çıktı şu şekilde olacaktır:

```
Üretici thread çalışıyor...
Tüketici thread çalışıyor...
Devam etmek için 'Enter'a basınız:
Tüketici devam ediyor...
Tüketici 5 saniye daha devam etti.
Üretici devam ediyor...
```

DİKKAT! Çıktıda önemli bir ayrıntı var: "notify()" metodu çağrılır çağrılmaz "wait()" ile beklemeye alınan thread devam etmiyor. Çünkü "consume()" metodu içerisindeki blok henüz tamamlanmadı ve dolayısıyla ilgili kilit serbest bırakılmadı. Çıktıdan da anlaşılacağı gibi 5 saniye geçtikten sonra kilit serbest kalıyor ve üretici çalışmasına devam edebiliyor.

DİKKAT! "wait()" metodu yalnızca "synchronized" kod blokları (yada "synchronized" metotlar) içerisinden çağrılabilir. Ve "wait()" metodu çağrılan nesne yalnızca söz konusu "synchronized" blokla ilişkili kilit nesnesi olabilir. Diyelim ki "synchronized" blok için belirtilen kilit nesnesi "lockObject" adında bir nesne olsun. Bu durumda bu blok içerisinde çağrılacak "wait()" metodu yalnızca "lockObject"e ait olan metot olabilir:

```
Object lockObject = new Object();
...
synchronized (lockObject) {
    ...
    lockObject.wait();
    ...
}
```

Örneğimizde kilit nesnesini geçerli nesne yani "this" olarak belirttiğimiz için blok içerisinde geçerli nesneye ait "wait()" metodunu çağırdık. Aksi halde "IllegalMonitorStateException" adlı istisna fırlatılırdı.

Bölüm 9: Yeniden Girilir Kilitler (Re-entrant Locks)

Bu bölümde "synchronized" anahtar kelimesinin işlevine alternatif bir yöntem olan yeniden girilir kilitleri (re-entrant locks) inceleyeceğiz.

Yeniden girilir kilitlerin çalışma mantığı şu şekildedir: java.util.concurrent.locks.* paketi altındaki "ReentrantLock" sınıfı türünde yeniden girilir bir kilit oluşturulur. Aynı anda yalnızca bir threadin girmesini istediğimiz kod bloğu yeniden girilir kilidin "lock()" ve "unlock()" metotlarıyla çevrelenir.

Şimdi örnek programımız üzerinden incelememize devam edelim:

```
public class Runner {

    private int count = 0;

    private Lock lock = new ReentrantLock();

    private void increment() {
        for (int i = 0; i < 10000; i++) {
            count++;
        }
    }

    public void firstThread() {
        lock.lock();
        increment();
        lock.unlock();
    }

    public void secondThread() {
        lock.lock();
        increment();
        lock.unlock();
    }

    public void printCount() {
        System.out.println("Sayaç: " + count);
    }

}
```

"Runner" sınıfımızın eş zamanlı iki ayrı thread içerisinde işletilecek iki metodu var: "firstThread()" ve "secondThread()". "main" metodumuz ise önceki bölümlerle benzer şeyi yapacak; bahsi geçen eş zamanlı iki threadi oluşturup başlatacak ve threadler sonlandığında "Runner" sınıfının "printCount()" metodu ile sayacın son durumunu yazdıracak. Bu nedenle "main" metodunu burada vermiyorum.

Bir thread "lock()" metodunu çağırarak o kilidi elde ettiğinde "lock()" çağrısı ile o kilidi elde etmek isteyen diğer threadler bekletilir. Kilidi elde etmiş thread içerisinde yeniden girilir kilidin "unlock()" metodu çağrılana kadar da bekletilmeye devam edilir. Böylece "increment()" metodu çağrıları sanki "synchronized" blok içerisindeymişcesine bir işlevsellik sağlanmış olur. Programı çalıştırdığımızda çıktı her seferinde şu şekilde olacaktır:

```
Sayaç: 20000
```

Diğer threadlerin kilidi elde edebilmesi için kilidi elde etmiş threadin kilidi serbest bırakması gerekir. Kilidi "ReentrantLock" nesnesine erişimi olan başka bir threadin serbest bırakması söz konusu değildir. Böyle bir durumda "IllegalMonitorStateException" istisnasının fırlatıldığını göreceksiniz. Dolayısıyla kilidi elde etmiş threadin bunu kesin olarak gerçekleştirmesi gerekmektedir. Peki ya çalışması sırasında bir istisna fırlatılır ve o thread henüz "unlock()" çağrısını yapamadan sonlanırsa? İşte bu nedenle "lock()" ve "unlock()" çağrıları arasında kalan kod bloğunun "try", "unlock()" çağrısının "finally" içerisine alınması yerinde bir pratik olacaktır. Böylece istisna fırlatılması halinde dahi kilit serbest bırakılmış olur:

```
public class Runner {

    ...

    public void firstThread() {
        lock.lock();
        try {
            increment();
        } finally {
            lock.unlock();
        }
    }

    public void secondThread() {
        lock.lock();
        try {
            increment();
        } finally {
            lock.unlock();
        }
```

```
    }

    ...

}
```

Yeniden girilir kilitlerin "synhronized" kod bloklarına bir alternatif olarak nasıl kullanılabileceklerini öğrenmiş olduk. Peki önceki bölümde incelediğimiz ve bize koşula göre bir threadi bekletme ve sonrasında sürdürme işlevselliğini kazandıran "wait()" ve "notify()" metotlarının bu alternatif yöntemdeki karşılığı nedir?

Bunun için "lock.newCondition()" çağrısından elde edilecek bir "java.util.concurrent.locks.Condition" nesnesine ihtiyacımız olacak. "Condition" sınıfının "await()" metodu "wait()" metoduna, "signal()" metodu ise "notify()" metoduna karşılık gelmektedir.

Şimdi "Condition" nesnesini ve bu metotları kullanarak "Runner" sınıfımızı önceki bölümde yer alan "Processor" sınıfına benzer bir yapıya kavuşturalım:

```
public class Runner {

    private int count = 0;

    private Lock lock = new ReentrantLock();

    private Condition condition = lock.newCondition();

    private void increment() {
        for (int i = 0; i < 10000; i++) {
            count++;
        }
    }

    public void firstThread() throws InterruptedException {
        lock.lock();

        System.out.println("Thread 1 çalışıyor...");

        condition.await();

        System.out.println("Thread 1 devam ediyor...");

        try {
            increment();
```

```
        } finally {
            lock.unlock();
        }
    }

    public void secondThread() throws InterruptedException {
        Thread.sleep(2000);

        lock.lock();

        System.out.println("Thread 2 çalışıyor...");
        System.out.print("Devam etmek için 'Enter'a basınız: ");

        new Scanner(System.in).nextLine();

        condition.signal();

        System.out.println("Thread 2 devam ediyor...");

        try {
            increment();
        } finally {
            lock.unlock();
        }
    }

    ...

}
```

Önce birinci threadin kilidi elde ettiğinden emin olmak için ikinci threadi 2 saniyeliğine bekletiyoruz. Programı çalıştırdığımızda çıktımız aşağıdaki şekilde olacaktır:

```
Thread 1 çalışıyor...
Thread 2 çalışıyor...
Devam etmek için 'Enter'a basınız:
Thread 2 devam ediyor...
Thread 1 devam ediyor...
Sayaç: 20000
```

Bölüm 10: Kilitlenme (Deadlock)

Bu bölümde çok threadli programların baş belası kilitlenme/tıkanma (deadlock) durumunu inceleyeceğiz.

Bu bölümdeki örnek programımızda iki banka hesabı arasındaki para transferini örnekleyeceğiz. Bir banka hesabını tanımlayan "Account" (Hesap) sınıfı aşağıdaki gibidir:

```
public class Account {

    private int balance = 10000;

    public static void transfer(Account sourceAccount, Account targetAccount,
                                int amount) {
        sourceAccount.withdraw(amount);
        targetAccount.deposit(amount);
    }

    public void deposit(int amount) {
        balance += amount;
    }

    public void withdraw(int amount) {
        balance -= amount;
    }

    public int getBalance() {
        return balance;
    }

}
```

Hesapların başlangıç bakiyesini 10 bin olarak ayarladık. "deposit" metodu bir hesaba para yatırmak için kullanılırken "withdraw" metodu o hesaptan para çekmek için kullanılacak. "transfer" yardımcı metodu ise belirtilen miktarda parayı ("amount" parametresi ile belirtiliyor) kaynak hesaptan çekip hedef hesaba yatırmada kullanılacak.

Şimdi para transferini gerçekleştirecek "Runner" sınıfımızı inceleyelim:

```
public class Runner {

    private Account account1 = new Account();

    private Account account2 = new Account();
```

```
    public void firstThread() {
        Random random = new Random();

        for (int i = 0; i < 10000; i++) {
            Account.transfer(account1, account2, random.nextInt(100));
        }
    }

    public void secondThread() {
        Random random = new Random();

        for (int i = 0; i < 10000; i++) {
            Account.transfer(account2, account1, random.nextInt(100));
        }
    }

    public void printBalance() {
        System.out.println("Hesap 1'in bakiyesi: " + account1.getBalance());
        System.out.println("Hesap 2'in bakiyesi: " + account2.getBalance());
        System.out.println("Toplam bakiye: "
                           + (account1.getBalance() + account2.getBalance()));
    }

}
```

"Runner" sınıfı, "main" metodunun eş zamanlı iyi ayrı thread içerisinde işleteceği iki metoda ("firstThread" ve "secondThread" metotları), bu iki thread sonlandığında "main" metodu tarafından çağrılarak hesap bakiyelerinin son durumlarını yazdıracak "printBalance()" metoduna ve son olarak iki adet hesap nesnesine ("Account" tipinde) sahip. "firstThread()" metodu döngü içerisinde belli miktar parayı (rastgele bir miktar) birinci hesaptan ikinci hesaba transfer ediyor. "secondThread()" metodu ise benzer para transferini ters yönde, yani ikinci hesaptan birinci hesaba olacak şekilde gerçekleştiriyor. Ve bakın çıktımız nasıl oluyor:

```
Hesap 1'in bakiyesi: 2905
Hesap 2'in bakiyesi: 14716
Toplam bakiye: 17621
```

Bazı durumlarda toplam bakiyeyi 20 bin olarak görebilirsiniz ancak çoğu durumda böyle alakasız sonuçlar görürsünüz. Bunun nedeni elbette ki eş zamanlı çalışan threadler. Çünkü iki thread de aynı anda birinci ve ikinci hesap bakiye değerleri (balance) üzerinde işlem yapmakta. Bu hatalı durumu çözmek için bir kilit mekanizması kullanmalı. Gelin geçen bölümde incelediğimiz yeniden girilir kilitlerden (re-entrant locks) kullanalım:

```
public class Runner {

    private Account account1 = new Account();
    private Account account2 = new Account();

    private Lock lock1 = new ReentrantLock();
    private Lock lock2 = new ReentrantLock();

    public void firstThread() {
        Random random = new Random();

        for (int i = 0; i < 10000; i++) {
            lock1.lock();
            lock2.lock();

            try {
                Account.transfer(account1, account2, random.nextInt(100));
            } finally {
                lock1.unlock();
                lock2.unlock();
            }
        }
    }

    public void secondThread() {
        Random random = new Random();

        for (int i = 0; i < 10000; i++) {
            lock1.lock();
            lock2.lock();

            try {
                Account.transfer(account2, account1, random.nextInt(100));
            } finally {
                lock1.unlock();
                lock2.unlock();
            }
        }
    }

    ...

}
```

Bir hesap üzerinde aynı anda yalnızca bir işlem yapılması gerektiği için hesap başına bir adet yeniden girilir kilit oluşturduk. Ayrıca bir transfer işlemi sırasında aynı anda iki hesap üzerinde

birden işlem yapıldığı için (birinden çekilip diğerine yatırılması şeklinde) transfer öncesi ilgili hesaplar için olan kilitlerin ikisi de kilitlenmeli ve işlem sonrasında iki kilit de serbest bırakılmalı.

Kilit mekanizması sayesinde artık çıktıyı her zaman toplam bakiye 20 bin olacak şekilde göreceğiz:

```
Hesap 1'in bakiyesi: 3985
Hesap 2'in bakiyesi: 16015
Toplam bakiye: 20000
```

Sorun giderildi. Yalnızca iki hesap olduğu için birer kilit oluşturup doğru sıralama ile "lock()" çağrılarını yaparak kolayca çözebildik. Peki "lock()" çağrılarının sıralamasını yanlış yapsaydık? Tıpkı aşağıdaki gibi:

```
public void firstThread() {
    ...

    for (int i = 0; i < 10000; i++) {
        lock1.lock();
        lock2.lock();

        ...
}

public void secondThread() {
    ...

    for (int i = 0; i < 10000; i++) {
        lock2.lock();
        lock1.lock();

        ...
}
```

Böyle bir durumda birinci thread birinci kilidi, ikinci thread ikinci kilidi elde edecek, ve aynı anda iki kilit de elde edildiği için kilitlenme yani "deadlock" meydana gelecekti. Böyle bir durumu önlemek için kullanılabilecek yöntemlerden biri "lock()" yerine "tryLock()" metodunu kullanmaktır. "tryLock()" metodu kilidi elde etmesi halinde "true" döner. Ancak belli süre geçtikten sonra hala kilidi elde edememişse bir istisna fırlatacaktır. Böylece biz durumdan haberdar oluruz ve belli süre sonra tekrar deneme gibi aksiyonlarda bulunabiliriz. Tıpkı aşağıdaki örneğimizde olduğu gibi:

```
public class Runner {

    ...

    private Lock lock1 = new ReentrantLock();
    private Lock lock2 = new ReentrantLock();

    private void acquireLocks(Lock firstLock, Lock secondLock) {
        while (true) {
            boolean isFirstLockAcquired = false;
            boolean isSecondLockAcquired = false;

            try {
                isFirstLockAcquired = firstLock.tryLock();
                isSecondLockAcquired = secondLock.tryLock();
            } finally {
                if (isFirstLockAcquired && isSecondLockAcquired) {
                    return;
                }

                if (isFirstLockAcquired) {
                    firstLock.unlock();
                }

                if (isSecondLockAcquired) {
                    secondLock.unlock();
                }
            }

            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
            }
        }
    }

    public void firstThread() {
        ...

        for (int i = 0; i < 10000; i++) {
            acquireLocks(lock1, lock2);

            ...
        }
    }

    public void secondThread() {
```

```
        ...

        for (int i = 0; i < 10000; i++) {
            acquireLocks(lock1, lock2);

            ...
        }
    }

    ...

}
```

"firstThread()" ve "secondThread()" metotları içerisindeki "lock()" çağrılarını kaldırarak yerine "acquireLocks" metodu çağrısını koyduk. "acquireLocks" metodu "tryLock()" metodunun sağladığı tekrar deneme yeteneğini kullanarak aynı anda iki kilit birden elde edilinceye kadar kilitleri elde etmeyi denemeye devam eder. Eğer kilitlerden yalnızca biri elde edilmişse kilitlenmeye (deadlock) sebebiyet vermemek için o kilidi serbest bırakır.

Bölüm 11: Semaforlar

Semaforlar (Semaphores) -genelde- bir kaynağa erişimi kontrol etmek için kullanılan sayaçlardır. Kilit mekanizmaları gibi threadlerin senkronizasyonu için kullanılırlar fakat kilitlerden farklı olarak bağımsız threadler tarafından serbest bırakılabilirler (release).

HATIRLATMA! Bir thread tarafından elde edilmiş kilidi başka bir thread serbest bırakmaya kalktığında "IllegalMonitorStateException" istisnası fırlatılır.

Örneğin programımızın bir sunucuya eş zamanlı isteklerde bulunması, ancak programımızdan o sunucuya eş zamanlı maksimum üç bağlantı olmasını istiyoruz. Tam da semaforlarla çözülebilecek bir durum. Kısıtlı kaynağımız "bağlantı sayısı", üst sınır değeri ise "3". Öyleyse 3 izinli (permit) bir semafor işimizi görecektir.

"java.util.concurrent.Semaphore" sınıfı kurucusu "permit" (izin) adlı bir parametre almaktadır. Bu değer ilgili kısıtın üst sınır değerini ifade eder. Örneğimiz için ifade edecek olursak, her bir thread bağlantı oluşturdukça semaforun "acquire()" (elde et) metodu ile semafordan bir izin alacaktır. İki thread daha bağlantı oluşturduğunda izin değeri "0" olacağından yeni bir thread izin istediğinde verilecek izin kalmadığından yeni threadler bekletilecektir. Ta ki önceki threadler işlerini bitirip -semaforun "release()" metodunu kullanarak- aldıkları izni iade edene kadar.

Örnek programımız sunucu bağlantısı kurmaya çalışan 10 adet threadi oluşturup başlatan bir "main" metodu ve kontrollü bir şekilde sunucu bağlantılarını gerçekleştiren bir "Connection" sınıfından meydana gelmekte:

```
public class Application {

    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newCachedThreadPool();

        for (int i = 0; i < 10; i++) {
            executorService.submit(() -> {
                try {
                    Connection.getInstance().connect();
                } catch (InterruptedException e) {
                }
            });
        }

        executorService.shutdown();
    }

}
```

"Connection" sınıfı:

```
public class Connection {

    private static Connection instance = new Connection();

    private int connectionCount = 0;

    private Semaphore semaphore = new Semaphore(3);

    private Connection() {
    }

    public static Connection getInstance() {
        return instance;
    }

    public void connect() throws InterruptedException {
        semaphore.acquire();

        try {
            doConnect();
        } finally {
            semaphore.release();
        }
    }

    private void doConnect() throws InterruptedException {
        synchronized (this) {
            connectionCount++;
            System.out.println("Bağlantı sayısı: " + connectionCount);
        }

        Thread.sleep(1000);

        synchronized (this) {
            connectionCount--;
        }
    }

}
```

"main" metodu bir thread havuzu oluşturarak içerisine aynı işi (sunucu bağlantısı kurmak) gerçekleştiren 10 adet thread ekler. Her biri "Connection" sınıfının "singleton" örneğinin "connect()" metodunu çağırmaktadır.

"connect()" metodunun başında "acquire()" metodu ile o thread için izin istenir. Eğer sayaç sıfıra ulaşmamışsa izin verilir ve bağlantı gerçekleştirilir. Bağlantı gerçekleştikten sonra alınan izin geri iade edilmelidir. Bunun için semaforun "release()" metodu kullanılır. Bağlantı sırasında hata alınması ihtimaline karşı try-finally bloğu kullanmak yerinde bir pratik olacaktır.

Programı çalıştırdığımızda çıktı aşağıdakine benzer şekilde olur:

```
Bağlantı sayısı: 1
Bağlantı sayısı: 2
Bağlantı sayısı: 3
Bağlantı sayısı: 1
Bağlantı sayısı: 2
Bağlantı sayısı: 3
Bağlantı sayısı: 3
Bağlantı sayısı: 2
Bağlantı sayısı: 3
Bağlantı sayısı: 2
```

İlk başta 3 bağlantı hemen kurulur. Fakat sonraki bağlantılar için önceki bağlantıların sonlanması, alınmış izinleri geri iade edilmesi gerekir. Semaforu oluştururken belirttiğimiz maksimum izin kısıtı (permit) sayesinde eş zamanlı bağlantı sayısı 3'ten fazla olamaz.

"Semaphore"un -boolean tipinde- ikinci bir parametre alan bir kurucusu daha mevcuttur. "fair" (adil) adlı bu "boolean" parametrenin değerinin "true" olması halinde izinlerin ilk giren ilk çıkar (first-in first-out) kuralıyla verileceği garanti edilir. Yani "acquire()" metodunu çağırarak izin isteyen bir threadin izni alan ilk thread olmasını garanti edilir. Performans gerekçeleriyle "fair" parametresinin varsayılan değeri "false"tur.

Bölüm 12: "Callable" ve "Future"

Thread havuzlarını incelediğimiz bölümde görmüştük: "ExecutorService" sınıfının "submit" metoduna yapılacak işleri birer "Runnable" nesnesi şeklinde geçeriz, o da her bir "Runnable" nesnesi için bir thread oluşturur ve başlatır. "submit" metodu eklediğimiz her bir "Runnable" için bir "Future" döndürür. Bu "Future" nesneleri sayesinde asenkron işletilen bu görevlerin tamamlanıp tamamlanmadığını, yoksa iptal mi edildiklerini kontrol edebilir, hatta o görevleri iptal edebiliriz. Ayrıca yine bu "Future" nesnelerini kullanarak threadlerin dönüş değerlerini elde edebiliriz. Şaşırmış olmalısınız, çünkü "Runnable" arabiriminin "run()" metodu değer döndürmez, yani "void"dir. Değer döndürme ihtiyacımız olduğu durumlarda Java eş zamanlılık kütüphanesinde yer alan "Callable" adlı arabirimi kullanabilir, "submit" metoduna görevlerimizi "Callable" nesneleri olarak geçebiliriz. "Callable" arabirimi jenerik tiplidir ve sahip olduğu tek metot olan "call()" metodu "Callable" jeneriği için belirtilen tipte bir değer döner.

Aşağıda "submit" metoduna rastgele bir sayı dönen bir "Callable" geçtik. Çıktı olaraksa "Future" nesnesi üzerinden elde ettiğimiz bu rastgele sayıyı yazdıracak:

```
public class Application {

    public static void main(String[] args) {
        ExecutorService executorService = Executors.newCachedThreadPool();

        Future<Integer> future = executorService.submit(new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                Random random = new Random();
                return random.nextInt();
            }
        });

        executorService.shutdown();

        try {
            Integer result = future.get();
            System.out.println("Sonuç: " + result);
        } catch (ExecutionException e) {
            System.out.println("Çalışma zamanı hatası!");
        } catch (InterruptedException e) {
        }
    }

}
```

Çıktı aşağıdakine benzer şekilde olur:

```
Sonuç: -574704917
```

"Future" sınıfının "get()" metodu iki tür "checked" ("catch" ile yakalanması gereken) istisna belirtir: "InterruptedException" ve "ExecutionException". İlki bir sonraki bölümde ele alacağımız threadin çalışmasının kesilmesi durumlarında fırlatılır. İkincisi yani "ExecutionException" ise "call()" metodu çalışırken herhangi bir istisna fırlatılması halinde fırlatılır.

"get()" metodu çağrıldığında threadin çalışması henüz sonlanmamışsa "get()" metodunu çağıran thread (örneğimiz için bu ana threaddir) bekletilir. "get()" metodunun zaman aşımı belirtebildiğimiz ikinci bir versiyonu daha vardır:

```
try {
    Integer result = future.get(10, TimeUnit.MILLISECONDS);
    System.out.println("Sonuç: " + result);
} catch (ExecutionException e) {
    System.out.println("Çalışma zamanı hatası!");
} catch (TimeoutException e) {
    System.out.println("Zaman aşımı!");
} catch (InterruptedException e) {
}
```

"get" metodunun bu versiyonuyla beraber ele almamız gereken istisnalara biri daha eklenir: "TimeoutException". İsminden de anlaşılacağı üzere thread belirtilen zamanda sonlanmazsa fırlatılır.

DİKKAT! Java 8 ve Lambda severler için belirtmekte fayda var. "Callable" arabirimi bir "@FunctionalInterface" olduğu için, "submit" metoduna "Callable" arabiriminin yapısına uygun bir Lambda geçilebilir:

```
Future<Integer> future = executorService.submit(() -> {
    Random random = new Random();
    return random.nextInt();
});
```

Bölüm 13: Threadlerin Yarıda Kesilmesi (Interrupting)

Bu zamana kadar ki örneklerimizde "InterruptedException" istisnası ile sıkça karşılaştık. Bu istisna bir threadin çalışması yarıda kesildiğinde fırlatılır. Peki çalışmakta olan bir threadin çalışması nasıl yarıda kesilir? Gelin yaptığı işlem uzun süren bir threadi belli süre geçtikten sonra durdurmayı deneyelim. Bunun için "Thread" sınıfının sağladığı "interrupt()" metodunu kullanacağız:

```
public class Application {

    public static void main(String[] args) throws InterruptedException {
        System.out.println("Başladı.");

        Thread t = new Thread(() -> {
            for (int i = 0; i < 1E8; i++) {
                Math.sin(new Random().nextDouble());
            }
        });

        t.start();

        Thread.sleep(500);

        t.interrupt();

        t.join();

        System.out.println("Bitti.");
    }

}
```

Programın çalışması bittiğinde çıktısının şu şekilde olduğunu göreceksiniz:

```
Başladı.
Bitti.
```

Benim gibi siz de "InterruptedException" istisnasının fırlatıldığını görmeyi beklemişsinizdir. Ancak gerçek şu ki "interrupt()" çağrısı yalnızca bir bayrağı (flag) bu threadin çalışmasının yarıda kesildiğini belirtecek şekilde ayarlar. Threadin çalışmasını durdurmak için döngü içerisinde "Thread.currentThread().isInterrupted()" kontrolünü yaparak döngüyü sonlandırmak bize kalmış durumda:

```
Thread t = new Thread(() -> {
    for (int i = 0; i < 1E8; i++) {
        if (Thread.currentThread().isInterrupted()) {
            System.out.println("Thread durduruldu!");
            break;
        }

        ...
    }
});
```

Programı bu değişiklik sonrasında çalıştırdığımızda çıktı şu şekilde olacaktır:

```
Başladı.
Thread durduruldu!
Bitti.
```

Diğer taraftan "run()" metodu içerisinde yapılacak "Thread.sleep()" ve "wait()" çağrılarında (ve "InterruptedException" fırlatabileceğini belirten diğer metotların çağrılarında) söz konusu bayrağın durumu kontrol edilerek otomatik olarak "InterruptedException" fırlatılır. Bu gibi bir durumda "catch" bloğu içerisinde gerekli gördüğümüz işlemleri yapabiliriz.