ÇOK KANALLI PROGRAMLAMA

Java, **çok kanallı programlamayı** (*multithreaded programming*) destekleyen bir dildir. Aynı anda çalışan birden fazla parçası olan programlara çok kanallı program denir. Bu parçaların her birine **iş parçacığı** (*thread*) denir.

Günümüz işlemcileri birden fazla iş parçacığını aynı anda çalıştırabilecek teknolojiye sahiptir. Bu teknolojiyi kullanarak, yazdığımız programları farklı iş parçacıklarına bölebilir ve daha verimli bir şekilde çalışmasını sağlayabiliriz.

Bugüne kadar gördüğümüz örnekler tek bir iş parçacığı olarak çalışıyordu. Bu iş parçacığına **ana iş parçacığı** (*main thread*) denir. Ana iş parçacığını başka iş parçacıkları oluşturmak için kullanabilirsiniz.

Aşağıdaki metodu inceleyelim:

```
public void run()
{
      a();
      b();
      c();
}
```

Yukarıda bir sınıfın içine yazılmış *run()* metodunu görüyorsunuz. Bu metot kendi içinde sırasıyla *a()*, *b()* ve *c()* metotlarını çağırıyor. Bu metotlar arka arkaya çağrıldıkları için sırayla çalıştırılırlar. İlk olarak *a()* metodu işletilir ve bitmesi beklenir, daha sonra *b()* metodu ve son olarak *c()* metodu çalıştırılır. Her metot çalışmak için bir önceki metodun bitmesini beklemelidir.

Şimdi kendimize şunu soralım: ya bu metotlar birbirinden tamamen bağımsızsa? Bu metotların birbirini hiçbir şekilde etkilemediğini, yaptıkları işlerin tamamen bir diğerinden bağımsız olduğunu düşünelim. Böyle bir durumda, bu metotların çalışması için bir öncekini beklemesine gerek yoktur. Her bir metot çalışmaya aynı anda başlayabilir. Bu durumda *run()* metodu daha verimli çalışmış olur.

İş parçacıklarını kullanarak birden fazla iş parçacığını aynı anda çalıştırabiliriz. Her işletim sisteminin, çalışmaya hazır iş parçacıklarını beklettiği bir liste ve bu listedeki iş parçacıklarını çalıştırmak üzere kendine özgü bir algoritması

vardır. Bu algoritmaya göre listeden bir iş parçacığı seçilir ve çalıştırılır. Burada şunu söylemek istiyoruz: bir iş parçacığı oluşturduğumuz zaman bunun ne zaman çalıştırılacağını önceden bilmemiz mümkün değildir; çünkü bu tamamen işletim sisteminin inisiyatifindedir.

Java'da hangi iş parçacığının daha önemli olduğunu belirlemek için bir öncelik değeri verilebilir. Bu değer iş parçacığının daha hızlı çalışmasını sağlamaz; fakat çalışacak bir sonraki iş parçacığını belirlerken etkili olabilir.

Bir iş parçacığının çalışmasının durdurulup başka bir iş parçacığının çalıştırılmasına **bağlam geçişi** (*context switch*) denir. Bağlam geçişinin iki sebebi olabilir:

- Çalışan bir iş parçacığı yaptığı işlemi tamamlamıştır. Bu durumda bu iş parçacığı sonlandırılır ve başka bir iş parçacığı çalıştırılır.
- Çalışan bir iş parçacığı işlemini tamamlamamış olsa bile daha yüksek önceliğe sahip başka bir iş parçacığının çalışması için durdurulabilir. Buna **kesme** (*interruption*) denir. Yüksek önceliğe sahip iş parçacıkları, daha düşük öncelikli iş parçacıklarını kesebilir.

Senkronizasyon

Çok kanallı programlama sayesinde birden fazla iş parçacığını aynı anda çalıştırabiliriz. Bu sayede sistemimizin kaynaklarını daha verimli kullanırız; fakat çoğu zaman iş parçacıkları arasındaki ilişkiyi düzenlememiz gerekir. İş parçacıkları asenkron olarak çalışır, yine de çoğu zaman birbirleriyle iletişime geçmeleri gerekir. Çoğu durumda, iş parçacıklarının birbirlerinin çalışmasını engellememeleri için düzenleme yapmamız gerekir. Bu düzenleme işlerine senkronizasyon denir. Örneğin, bir iş parçacığı bir dosya üzerinde okuma yapıyorsa, başka bir iş parçacığının bu dosyaya yazmasını engellemek isteyebilirsiniz. Java, senkronizasyonu dile entegre bir biçimde destekler. Bununla ilgili örnekleri daha sonra göreceğiz.

Thread sınıfı

Java'nın çok kanallı programlama mimarisi *Thread* sınıfı üzerine kurulmuştur.

İş parçacıklarını bu sınıfı kullanarak temsil ederiz. Yeni bir iş parçacığı oluşturmak için ya *Thread* sınıfının bir alt sınıfını oluşturur ya da *Runnable* arayüzünü kullanırız.

Thread sınıfının bazı metotlarını inceleyelim:

String getName()	İş parçacığının ismini döndürür. *setName()* metodunu kullanarak iş parçacığına bir isim verebilirsiniz.
int getPriority()	İş parçacığının öncelik değerini döndürür. Bu değer 1 ile 10 arasında bir tam sayıdır.
boolean isAlive()	İş parçacığının hâlâ çalışıp çalışmadığı bilgisini döndürür.
void join()	Başka bir iş parçacığının tamamlanmasını bekler.
void sleep(**long** millis)	İş parçacığının çalışmasını belirtilen milisaniye kadar durdurur.
void start()	İş parçacığını başlatır.
static Thread currentThread()	Şu an çalışmakta olan iş parçacığını döndürür.

InterruptedException

Bir iş parçacığının çalışmasının başka bir iş parçacığı tarafından kesilme ihtimali her zaman vardır. Bu durum meydana geldiğinde *InterruptedException* hatası fırlatılır. Java'da iş parçacıkları üzerinde işlem yaparken her zaman try-catch bloğu kullanmalıyız.

İş parçacığı oluşturmak

Java'da yeni bir iş parçacığı oluşturmak için 2 farklı yöntem kullanabilirsiniz:

- *Runnable* arayüzünü uygulayan bir sınıf oluşturabilirsiniz.
- *Thread* sınıfının alt sınıfını oluşturabilirsiniz.

Runnable arayüzünü kullanarak iş parçacığı oluşturmak

Runnable arayüzünün tek bir metodu vardır:

Runnable	
void run()	İş parçacığının yapacağı işlemleri bu metot içine yazarız.

Şimdi aşağıdaki örnek sınıfı inceleyelim:

```
public class MyNewThread implements Runnable
{

    private final Thread thread;

    public MyNewThread()
    {
        thread = new Thread(this, "Yeni-thread'imiz");
    }

    public void start()
    {
        thread.start();
    }

    @Override
    public void run()
    {
        try
        {
            System.out.println(thread.getName() +
                  " çalışmaya başladı.");
            System.out.println(thread.getName() +
                  " 2 saniye bekletiyorum...");
            Thread.sleep(2000L);
            System.out.println(thread.getName() +
                  " çalışması tamamlandı.");
        }
        catch (InterruptedException ex)
        {
            System.err.println(thread.getName() +
                  " çalışması durduruldu!");
        }
    }

}
```

Yukarıda, *Runnable* arayüzünü uygulayan yeni bir sınıf oluşturduk. Bu sınıfın yapılandırıcısında yeni bir *Thread* oluşturduk ve parametre olarak bir isim

verdik. *Thread* sınıfının bir örneğini alırken parametre olarak *Runnable* türünde bir nesne vermelisiniz.

Runnable arayüzünü uyguladığımız için *run()* metodu yazdık. Bu metodun içinde, yeni bir iş parçacığı olarak çalışacak kodları yazdık. Burada iş parçacığını 2 saniye beklettik. Ayrıca kodları try-catch bloğu içinde yazdığımıza dikkatinizi çekerim.

Son olarak, iş parçacığını başlatabilmek için *MyNewThread* isimli sınıfımıza *start()* metodunu ekledik.

Şimdi bu iş parçacığını çalıştıralım:

```
MyNewThread thread = new MyNewThread();
thread.start();
```

Yukarıdaki kodu çalıştırdığınız zaman çıktısı aşağıdaki gibi olur:

```
Yeni-thread'imiz çalışmaya başladı.
Yeni-thread'imiz 2 saniye bekletiyorum...
Yeni-thread'imiz çalışması tamamlandı.
```

Bu örnekten iş parçacıklarının çalışma prensibini tam olarak anlayamayız; çünkü gördüğünüz gibi, yalnızca *run()* metodunun çıktısını görüyoruz. Şimdi başka bir örnek yapalım ve birden fazla iş parçacığını aynı anda çalıştıralım:

```
public class CountingThread implements Runnable
{

    private final String name;

    public CountingThread(String name)
    {
        this.name = name;
        new Thread(this, name).start();
    }

    @Override
    public void run()
    {
        for (int i = 1; i <= 10; i++)
        {
            System.out.println(name + " - " + i);
        }
    }

}
```

Yukarıda, *CountingThread* isminde, *Runnable* arayüzünü uygulayan bir sınıf yazdık. Bu sınıfı yeni bir iş parçacığı oluşturmak için kullanacağız. Bu iş parçacığında yaptığımız ise basit: 1'den 10'a kadar olan sayıları iş parçacığının ismiyle birlikte konsola yazdırıyoruz.

Şimdi bu sınıfı kullanarak 2 farklı iş parçacığı oluşturalım:

```
CountingThread thread1 = new CountingThread("A");
CountingThread thread2 = new CountingThread("B");
```

Yukarıdaki kodu çalıştırdığımda aşağıdaki çıktıyı elde ettim:

```
B - 1
A - 1
B - 2
A - 2
B - 3
A - 3
B - 4
A - 4
B - 5
A - 5
B - 6
A - 6
B - 7
A - 7
B - 8
A - 8
B - 9
A - 9
B - 10
A - 10
```

Gördüğünüz gibi, iş parçacıklarımız aynı anda çalışmışlar, dolayısıyla sanki konsola sırayla yazıyorlar gibi gözükmüştür. Bu programı her çalıştırdığınızda çıktısı farklı olacaktır.

Thread sınıfını genişleterek iş parçacığı oluşturmak

Yukarıda yaptığımız son örneği aşağıdaki gibi yapabiliriz:

```
public class CountingThread extends Thread
{

    public CountingThread(String name)
    {
        super(name);
    }

    @Override
    public void run()
    {
        for (int i = 1; i <= 10; i++)
        {
            System.out.println(getName() + " - " + i);
        }
    }

}
```

Bir önceki örnekle arada bazı farklar vardır. Öncelikle, iş parçacığının ismini parametre olarak aldık ve üst sınıfın (*Thread* sınıfının) yapılandırıcısına parametre olarak verdik. Artık iş parçacığının ismini *name* adında bir değişkende tutmamıza gerek yoktur; çünkü *getName()* metoduyla erişebiliriz.

Yine de, *run()* metodunun varlığını koruduğuna dikkat edin. Fakat, bu *run()* metodu *Runnable* arayüzünden değil, *Thread* sınıfından geliyor. Eğer *Thread* sınıfını genişleterek iş parçacığı oluşturacaksanız, *run()* metodunu ezmeniz gerekir.

Bu örneği şu şekilde çalıştırabiliriz:

```
CountingThread thread1 = new CountingThread("A");
CountingThread thread2 = new CountingThread("B");
thread1.start();
thread2.start();
```

Senkronize metotlar (Synchronized methods)

Birden fazla iş parçacığını aynı anda çalıştırmak çoğu zaman faydalıdır. Fakat bazen, bir metodu birden fazla iş parçacığının aynı anda çalıştırmamasını istersiniz. Bunun için metoda **kilit** (*lock*) koymak gerekir. Kilit koyulan bir metodu bir iş parçacığı çalıştırdığı zaman kilitlemiş olur, bu süre boyunca başka bir iş parçacığı bu metodu çalıştıramaz. Metodu çalıştıran iş parçacığı metottan çıktıktan sonra kilit kaldırılmış olur, artık başka iş parçacıkları da bu metodu çalıştırabilir.

Java'nın güzel özelliklerinden biri de kilit mekanizmasını doğrudan destekliyor olmasıdır. Bir metoda kilit koymak istiyorsak bunu **synchronized** deyimiyle yaparız. Senkronize olarak tanımlanmış bir metodu aynı anda en fazla 1 iş parçacığı çalıştırabilir. Şimdi bununla ilgili bir örnek yapalım.

Öncelikle, *CommonObject* isminde bir sınıf tanımlayalım:

```
public class CommonObject
{

    public void execute()
    {
        Thread currentThread = Thread.currentThread();

        System.out.println(currentThread.getName() + " çalışıyor...");
        System.out.println(currentThread.getName() +
            " çalışmasını   tamamladı.");
    }

}
```

Bu sınıfın içine *execute()* adında bir metot yazdık. Bu metot o anda çalışan iş parçacığının ismini alıp konsola yazdıracak.

Şimdi iş parçacığı oluşturmak için *MyThread* adında bir sınıf oluşturalım:

```
public class MyThread extends Thread
{

    private CommonObject commonObject;

    public MyThread(String threadName)
    {
        super(threadName);
    }

    public void start(CommonObject commonObject)
    {
        this.commonObject = commonObject;
        super.start();
    }

    @Override
    public void run()
    {
        System.out.println(getName() + " bekliyor...");
        commonObject.execute();
    }

}
```

Bu sınıfı iş parçacığı oluşturmak için kullanacağız. Sınıfın bir örneğini alırken parametre olarak bir isim veriyoruz. Bu isim, iş parçacığının ismi olacak. İş parçacığını başlatırken parametre olarak *CommonObject* türünde bir nesne veriyoruz. İş parçacığı çalışırken bu nesnenin yukarıda yazdığımız execute() metodunu çağıracak.

Şimdi bu sınıfları kullanalım. 10 tane farklı iş parçacığı oluşturalım ve *CommonObject* sınıfını *execute()* metodunun nasıl çağrıldığını görelim:

```
CommonObject commonObject = new CommonObject();

for (int i = 1; i <= 10; i++)
{
    MyThread thread = new MyThread(String.valueOf(i));
    thread.start(commonObject);
}
```

Yukarıdaki kodu çalıştırdığım zaman aşağıdaki çıktıyı aldım (bu çıktı her zaman farklı olur):

```
9 bekliyor...
3 bekliyor...
1 bekliyor...
10 bekliyor...
4 bekliyor...
6 bekliyor...
2 bekliyor...
4 çalışıyor...
7 bekliyor...
5 bekliyor...
7 çalışıyor...
2 çalışıyor...
1 çalışıyor...
9 çalışıyor...
10 çalışıyor...
6 çalışıyor...
9 çalışmasını tamamladı.
1 çalışmasını tamamladı.
8 bekliyor...
2 çalışmasını tamamladı.
6 çalışmasını tamamladı.
7 çalışmasını tamamladı.
10 çalışmasını tamamladı.
4 çalışmasını tamamladı.
3 çalışıyor...
5 çalışıyor...
3 çalışmasını tamamladı.
8 çalışıyor...
5 çalışmasını tamamladı.
8 çalışmasını tamamladı.
```

Gördüğünüz gibi, oluşturduğumuz 10 farklı iş parçacığının her biri aynı anda *execute()* metodunu çalıştırabilmiştir. Fakat biz, *execute()* metodunun aynı anda en fazla 1 iş parçacığı tarafından çalıştırılmasını istiyoruz. Şimdi bunun

için *execute()* metodunu yeniden tanımlayalım ve **synchronized** deyimini ekleyelim:

```
public class CommonObject
{

    public synchronized void execute()
    {
        Thread currentThread = Thread.currentThread();

        System.out.println(currentThread.getName() + " çalışıyor...");
        System.out.println(currentThread.getName() +
             " çalışmasını tamamladı.");
    }

}
```

Artık *execute()* metodumuzu senkronize bir metot olarak tanımladık. Bundan böyle aynı anda en fazla 1 iş parçacığı tarafından çalıştırılabilecektir. Şimdi yukarıdaki kodu tekrar çalıştırdığımız zaman şuna benzer bir çıktı alırız:

```
6 bekliyor...
4 bekliyor...
2 bekliyor...
1 bekliyor...
9 bekliyor...
3 bekliyor...
5 bekliyor...
7 bekliyor...
8 bekliyor...
10 bekliyor...
6 çalışıyor...
6 çalışmasını tamamladı.
10 çalışıyor...
10 çalışmasını tamamladı.
8 çalışıyor...
8 çalışmasını tamamladı.
7 çalışıyor...
7 çalışmasını tamamladı.
5 çalışıyor...
5 çalışmasını tamamladı.
3 çalışıyor...
3 çalışmasını tamamladı.
9 çalışıyor...
9 çalışmasını tamamladı.
1 çalışıyor...
1 çalışmasını tamamladı.
4 çalışıyor...
4 çalışmasını tamamladı.
2 çalışıyor...
2 çalışmasını tamamladı.
```

İki çıktı arasındaki farkı kolaylıkla görebilirsiniz. Artık *execute()* metodu senkronize olduğu için, aynı anda 1 iş parçacığı tarafından çalıştırılabilmektedir.

Senkronize bloklar (Synchronized blocks)

Yukarıda bir metodu nasıl senkronize yapacağımızı gördük. Bu işlem yararlı olsa da bazen bir metodun tamamını senkronize olarak belirtmek programın performansını düşürebilir. Bir metodun içinde, farklı iş parçacıkları tarafından çalıştırıldığında sorun teşkil edecek ve etmeyecek farklı kısımlar bulunabilir. Programcılar olarak bu kısımları doğru belirlemek bizim sorumluluğumuzdur.

Bir algoritmanın, aynı anda en fazla 1 tane iş parçacığı tarafından çalıştırılması gereken kısmına **kritik bölge** (*critical section*) denir. Bu bölgeye kritik deriz;

çünkü aynı anda birden fazla iş parçacığı tarafından çalıştırılırsa istenmeyen durumlar oluşabilir.

Yazdığımız bir metodun içindeki yalnızca kritik bölgeye kilit koymak istiyorsak, bunu senkronize bloklar sayesinde yaparız.

Bir metodu senkronize yaptığımız zaman o metoda kilit koyulur. Fakat bir bloğu senkronize yapmak istersek kilit olarak kullanacağımız bir nesneye ihtiyacımız vardır. Bu nesne bu amaçla kullanılacağı için, başka amaçlarla kullanılmaması ve değiştirilmemesi (**final** olması) gerekir.

Senkronize blokların yapısı şöyledir:

```
synchronized ( [kilit olarak kullanılacak nesne] )
{
        [kritik bölge kodları]
}
```

Senkronize blok oluşturmak için **synchronized** deyimiyle yeni bir blok açılır ve kilit olarak kullanılacak nesne parantez içinde belirtilir. Bu blok içinde yazacağımız kodlar aynı anda en fazla 1 iş parçacığı tarafından çalıştırılır.

Şimdi, yukarıdaki örneği değiştirelim ve senkronize blok kullanacak şekilde tekrar yazalım:

```
public class CommonObject
{

    private final Object lock = new Object();

    public void execute()
    {
        Thread currentThread = Thread.currentThread();
        System.out.println(currentThread.getName() + " bekliyor...");

        synchronized (lock)
        {
            System.out.println(currentThread.getName() +
                  " çalışıyor...");
            System.out.println(currentThread.getName() +
                  " çalışmasını tamamladı.");
        }
    }

}
```

CommonObject sınıfındaki farkları inceleyelim. Öncelikle, artık *execute()* metodunun tamamının senkronize olmadığını görüyoruz. Yalnızca kritik bölge olarak seçtiğimiz kısmı senkronize blok içine yazdık. Ayrıca bu blokta kullanmak üzere bir kilit nesnesi oluşturduk. Bu nesnenin sabit (final) olduğunu da belirtelim.

```
public class MyThread extends Thread
{

    private CommonObject commonObject;

    public MyThread(String threadName)
    {
        super(threadName);
    }

    public void start(CommonObject commonObject)
    {
        this.commonObject = commonObject;
        super.start();
    }

    @Override
    public void run()
    {
        commonObject.execute();
    }

}
```

MyThread sınıfında yaptığımız değişiklik, iş parçacığın metodu çalıştırmayı beklediğini konsola yazdıran satırı silmekten ibarettir; çünkü bu kısmı *execute()* metodunun içine taşıdık.

```
CommonObject commonObject = new CommonObject();

for (int i = 1; i <= 10; i++)
{
    MyThread thread = new MyThread(String.valueOf(i));
    thread.start(commonObject);
}
```

Şimdi yukarıdaki kodu çalıştırdığımız zaman aşağıdakine benzer bir çıktı alırız:

```
3 bekliyor...
5 bekliyor...
7 bekliyor...
8 bekliyor...
9 bekliyor...
1 bekliyor...
6 bekliyor...
10 bekliyor...
4 bekliyor...
3 çalışıyor...
2 bekliyor...
3 çalışmasını tamamladı.
2 çalışıyor...
2 çalışmasını tamamladı.
4 çalışıyor...
4 çalışmasını tamamladı.
10 çalışıyor...
10 çalışmasını tamamladı.
5 çalışıyor...
5 çalışmasını tamamladı.
6 çalışıyor...
6 çalışmasını tamamladı.
1 çalışıyor...
1 çalışmasını tamamladı.
9 çalışıyor...
9 çalışmasını tamamladı.
8 çalışıyor...
8 çalışmasını tamamladı.
7 çalışıyor...
7 çalışmasını tamamladı.
```

Gördüğünüz gibi, metodu aynı anda birden fazla iş parçacığı çalıştırabilmiş; fakat senkronize blok içine en fazla 1 tanesi aynı anda girebilmiştir.

Başka bir iş parçacığının çalışmasını beklemek

Bir iş parçacığını başlattığınızda ne zaman çalışacağını önceden kestiremezsiniz. Fakat *Thread* sınıfının *join()* metodunu kullanarak bu iş parçacığının çalışmasının bitmesini bekleyebilirsiniz. Bunu bir örnekle görelim:

```
Thread t =  new Thread(() -> {
    for (int i = 1; i <= 10; i++)
    {
        System.out.println("Thread içinde yazıyorum: " + i);
    }
});

t.start();

try
{
    Thread.sleep(10L);
}
catch (InterruptedException ex)
{
    throw new RuntimeException(ex);
}
finally
{
    System.out.println("main() metodunun sonuna geldik.");
}
```

Yukarıda yeni bir iş parçacığı oluşturuyor ve 1'den 10'a kadar olan sayıları konsola yazdırıyoruz. Fakat bu iş parçacığını başlattığımız zaman, artık elimizde çalışan 2 kod vardır. Bir yandan bu iş parçacığı çalışmakta, diğer yandan *main()* metodu çalışmasına devam etmektedir. Bu kodu çalıştırdığım zaman şöyle bir çıktı aldım:

```
Thread içinde yazıyorum: 1
main() metodunun sonuna geldik.
Thread içinde yazıyorum: 2
Thread içinde yazıyorum: 3
Thread içinde yazıyorum: 4
Thread içinde yazıyorum: 5
Thread içinde yazıyorum: 6
Thread içinde yazıyorum: 7
Thread içinde yazıyorum: 8
Thread içinde yazıyorum: 9
Thread içinde yazıyorum: 10
```

Gördüğünüz gibi, iş parçacığı tamamlanmadan *main()* metodu sonlanmıştır. Eğer "main() metodunun sonuna geldik." yazısının en son satırda yazmasını istiyorsak oluşturduğumuz iş parçacığının tamamlanmasını beklememiz gerekir. Bunu aşağıdaki gibi yapabiliriz:

```
Thread t =  new Thread(() -> {
    for (int i = 1; i <= 10; i++)
    {
        System.out.println("Thread içinde yazıyorum: " + i);
    }
});

t.start();

try
{
    t.join();
    Thread.sleep(10L);
}
catch (InterruptedException ex)
{
    throw new RuntimeException(ex);
}
finally
{
    System.out.println("main() metodunun sonuna geldik.");
}
```

Gördüğünüz gibi, *join()* metodunu kullanarak iş parçacığının bitmesini bekliyoruz. Artık bu kodu çalıştırdığınız zaman *main()* metodu *t* iş parçacığını bekleyecektir:

```
Thread içinde yazıyorum: 1
Thread içinde yazıyorum: 2
Thread içinde yazıyorum: 3
Thread içinde yazıyorum: 4
Thread içinde yazıyorum: 5
Thread içinde yazıyorum: 6
Thread içinde yazıyorum: 7
Thread içinde yazıyorum: 8
Thread içinde yazıyorum: 9
Thread içinde yazıyorum: 10
main() metodunun sonuna geldik.
```

Çıkmaz (*deadlock*)

Birden fazla iş parçacığıyla uğraşırken kodunuzu çok dikkatli yazmalısınız. Çok kanallı programlama dikkatli yapılmazsa birçok hataya sebep olabilir. Bu hataların en bilineni çıkmazdır. İki iş parçacığının birbirini beklemesine **çıkmaz** (*deadlock*) denir. Bu istenmeyen bir durumdur; çünkü iki iş parçacığı da çalışmak için bir diğerini bekleyecek ve çalışamayacaktır. Bu durum iki iş

parçacığının çıkmaza girmesine ve sonsuza kadar birbirini beklemesine neden olur. Örneğin, aşağıdaki kodu inceleyelim:

```
public class MyThread extends Thread
{

    private Thread otherThread = null;

    public void execute(Thread otherThread)
    {
        this.otherThread = otherThread;
        start();
    }

    @Override
    public void run()
    {
        try
        {
            if (otherThread != null)
            {
                System.out.println(getName() + ", " +
                     otherThread.getName() +
                     " iş parçacığını bekliyor...");

                otherThread.join();
            }

            System.out.println(getName() + " işini tamamladı.");
        }
        catch (InterruptedException ex)
        {
            throw new RuntimeException(ex);
        }
    }

}
```

Burada, *MyThread* isminde bir *Thread* sınıfı oluşturduk. Bu iş parçacığını başlatırken parametre olarak başka bir *Thread* nesnesi alıyoruz ve **null** değilse bu nesnenin çalışmasını bekliyoruz.

Şimdi aşağıdaki kodu inceleyelim:

```
MyThread thread1 = new MyThread();
MyThread thread2 = new MyThread();

thread1.setName("A");
thread2.setName("B");

thread1.execute(thread2);
thread2.execute(thread1);
```

Bu kodu çalıştırdığınız zaman aşağıdaki çıktıyı alırsınız:

```
B, A iş parçacığını bekliyor...
A, B iş parçacığını bekliyor...
```

Bu çıktıyı almanıza rağmen program sonlanmaz, çünkü A ve B iş parçacıkları birbirini beklemeye başlar. Bu durum, siz programı sonlandırmadığınız sürece sonsuza dek sürer. Çıkmaz (*deadlock*), kötü programlama örneğidir ve buna göre mutlaka tedbir almanız gerekir.

İş parçacığı durumları (Thread states)

Thread sınıfı içinde *State* adında bir **enum** tanımlanmıştır. Bir iş parçacığının o anki durumunu öğrenmek için bu **enum** değerlerini kullanırız. *Thread* sınıfının *getState()* metodu bu **enum** türünde bir değer döndürür ve bize iş parçacığının o anki durumu hakkında bilgi verir. Bu **enum** değerleri aşağıdaki gibidir:

NEW	İş parçacığı yeni oluşturulmuş ve henüz çalışmaya başlamamış
RUNNABLE	İş parçacığı şu anda çalışıyor veya işletim sistemi tarafından çalışma sırasına alınmış
BLOCKED	İş parçacığı çalışmıyor; çünkü senkronize bir metot veya blok içinde bekliyor
WAITING	İş parçacığı çalışmıyor; çünkü başka bir iş parçacığını bekliyor
TIMED_WAITING	İş parçacığı çalışmıyor; çünkü belli bir süre çalışmaması istenmiş
TERMINATED	İş parçacığı çalışmış ve işlemini tamamlamış

Çalışmış ve işini tamamlamış (*TERMINATED* durumundaki) bir *Thread* nesnesini tekrar çalıştıramazsınız. Bunun için yeni bir *Thread* nesnesi oluşturmanız gerekir.

PARALEL PROGRAMLAMA

Bir önceki başlıkta iş parçacığı kavramından ve nasıl iş parçacığı oluşturulduğundan bahsettik. Ayrıca, iş parçacıklarıyla ilgili bazı önemli kavramları not ettik. Bunları çok kanallı programlama kavramıyla ifade ettik.

Bu başlıkta ise iş parçacıkları arasındaki iletişimi ve senkronizasyonu sağlayan bazı gelişmiş Java sınıflarından bahsedeceğiz. Bu yüzden bunları paralel programlama başlığı altında topladık.

Senkronizasyon nesneleri

İlk olarak senkronizasyon nesnelerini inceleyeceğiz. Bu nesneler, iş parçacıkları arasındaki çalışma sıralaması ve bilgi alışverişi gibi işlemlerde gelişmiş yöntemler sunar. Bu başlık altında 5 farklı senkronizasyon nesnesini inceleyeceğiz. Bu sınıflar java.util.concurrent pakedi altındadır:

- Semaphore
- CountDownLatch
- CyclicBarrier
- Exchanger

Semaphore

En bilindik senkronizasyon aracı semafordur. Semafor, en basit ifadesiyle, birden fazla iş parçacığına izin veren bir kilittir. Daha önce gördüğümüz gibi, kilitler aynı anda en fazla 1 tane iş parçacığının çalışmasına izin veriyordu. Semaforlar da bu mekanizmayla çalışır, yalnızca izin verilen iş parçacığı sayısı 1'den fazladır.

Semafor oluştururken, izin vereceğimiz iş parçacığı sayısını parametre olarak veririz. Bu sayı sıfırdan büyük olmalıdır.

Semafor kendi içinde bir sayaç tutar. Belli sayıda iş parçacığıyla kısıtlamak istediğimiz kodun başında semaforun *acquire()* metodunu çağırırız. Böylece, semafordan izin istemiş oluruz. Eğer başlangıçta belirttiğimiz kadar iş parçacığı çalışmıyorsa, semafor bize izin verir ve sayacı bir azaltır. Sayaç sıfıra vurursa semafor gelen isteklere izin vermez ve onları bekletir.

İşimiz bittiğinde semaforun *release()* metodunu çağırırız. Böylece semafor sayacı bir artırır ve bekleyen bir iş parçacığı varsa devam etmesi için ona izin verir.

Şimdi, daha önce yaptığımız örneğin bir benzerini yapalım:

```
public class CommonObject
{

    public void execute()
    {
        Thread currentThread = Thread.currentThread();
        System.out.println(currentThread.getName() + " çalışıyor...");
        System.out.println(currentThread.getName() +
             " çalışmasını tamamladı.");
    }

}
```

CommonObject sınıfımızı ve *execute()* metodunu hatırlamışsınızdır. Gördüğünüz gibi metodu senkronize olarak belirtmedik ve senkronize blok yazmadık.

```
import java.util.concurrent.Semaphore;

public class MyThread extends Thread
{

    private final CommonObject commonObject;
    private final Semaphore semaphore;

    public MyThread(String threadName, CommonObject commonObject,
             Semaphore semaphore)
    {
        super(threadName);
        this.commonObject = commonObject;
        this.semaphore = semaphore;
    }

    @Override
    public void run()
    {
        try
        {
            semaphore.acquire();
            commonObject.execute();
        }
        catch (InterruptedException ex)
        {
            System.err.println(getName() + " kesildi!");
        }
        finally
        {
            semaphore.release();
        }
    }

}
```

MyThread sınıfını da yukarıdaki gibi güncelledik. Sınıfı oluştururken *CommonObject* ve *Semaphore* türünde iki nesne alıyoruz. Bu iş parçacığı çalışırken öncelikle *acquire()* metoduyla semafordan izin istiyoruz. Semafor izin verirse *commonObject* nesnesinin *execute()* metodunu çağırıyoruz. *acquire()* metodu bize her zaman izin vermeyebilir ve bizi bekletebilir; bu durumda *InterruptedException* fırlatma ihtimali vardır. Buna karşı önlemimizi alıyoruz. Son olarak, *release()* metodunu kullanarak semafora işimizin bittiğini bildiriyoruz. Burada dikkat etmeniz gereken bir nokta var: semafor kullanırken *release()* metodunu her zaman **finally** bloğu içinde çağırmalısınız! Bu sayede sayacın düzgün bir şekilde çalışmasını garanti altına almış oluruz.

Şimdi bu sınıfları aşağıdaki gibi kullanalım:

```
CommonObject commonObject = new CommonObject();
Semaphore semaphore = new Semaphore(3);

for (int i = 1; i <= 12; i++)
{
    MyThread thread = new MyThread(String.valueOf(i),
          commonObject, semaphore);
    System.out.println(i + " bekliyor...");
    thread.start();
}
```

Gördüğünüz gibi, semaforu oluştururken parametre olarak 3 verdik. Bu yüzden semafor aynı anda en fazla 3 iş parçacığına izin verecektir. Bu kodu çalıştırdığınız zaman aşağıdakine benzer bir çıktı alırsınız:

```
1 bekliyor...
2 bekliyor...
3 bekliyor...
4 bekliyor...
5 bekliyor...
6 bekliyor...
7 bekliyor...
8 bekliyor...
9 bekliyor...
10 bekliyor...
11 bekliyor...
12 bekliyor...
4 çalışıyor...
3 çalışıyor...
2 çalışıyor...
4 çalışmasını tamamladı.
7 çalışıyor...
2 çalışmasını tamamladı.
3 çalışmasını tamamladı.
6 çalışıyor...
7 çalışmasını tamamladı.
6 çalışmasını tamamladı.
8 çalışıyor...
5 çalışıyor...
1 çalışıyor...
5 çalışmasını tamamladı.
8 çalışmasını tamamladı.
9 çalışıyor...
1 çalışmasını tamamladı.
9 çalışmasını tamamladı.
10 çalışıyor...
12 çalışıyor...
11 çalışıyor...
12 çalışmasını tamamladı.
10 çalışmasını tamamladı.
11 çalışmasını tamamladı.
```

Çıktıyı incelerseniz fark edeceksiniz, aynı anda en fazla 3 iş parçacığı işlemini yapmaktadır.

CountDownLatch

Bazen, belli sayıda iş parçacığı çalışana kadar başka bir iş parçacığını bekletmek istersiniz. Bu gibi durumlarda *CountDownLatch* kullanılır. Bu nesneyi geri sayım sayacı olarak düşünebilirsiniz. Nesneyi oluştururken parametre olarak bir sayı verirsiniz ve her iş parçacığının sonunda bu sayıyı bir azaltırsınız. Sayaç sıfıra vurduğunda beklettiğiniz iş parçacığına çalışma izni verilir.

CountDownLatch şu şekilde kullanılır:

- Nesneyi oluştururken yapılandırıcısına parametre olarak beklemek istediğimiz iş parçacığı sayısını veririz.
- Bekletmek istediğimiz iş parçacığının içinde *await()* metodunu çağırırız.
- Diğer iş parçacıklarında ise işlem tamamlandığı zaman *countDown()* metodunu çağırırız.

Aşağıdaki örneği inceleyelim:

```
import java.util.concurrent.CountDownLatch;

public class MyThread extends Thread
{

    private final CountDownLatch latch;

    public MyThread(CountDownLatch latch, String name)
    {
        this.latch = latch;
        setName(name);
    }

    @Override
    public void run()
    {
        try
        {
            System.out.println(getName() + " çalışıyor...");
        }
        catch (Exception ex)
        {
            throw new RuntimeException(ex);
        }
        finally
        {
            latch.countDown();
        }
    }

}
```

MyThread isminde bir *Thread* tanımladık. Bu nesneyi oluştururken parametre olarak bir *CountDownLatch* nesnesi alıyoruz. *Thread* çalıştıktan sonra bu sayacı *countDown()* metoduyla bir azaltıyoruz. Dikkat etmeniz gereken nokta şudur: *countDown()* metodunu her zaman **finally** bloğu içinde çağırmalısınız.

```
import java.util.concurrent.CountDownLatch;

public class WaitingThread extends Thread
{

    private final CountDownLatch latch;

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

    @Override
    public void run()
    {
        try
        {
            System.out.println("Bekleyen thread bekliyor...");

            latch.await();

            System.out.println("Bekleyen thread çalıştı.");
        }
        catch (InterruptedException ex)
        {
            throw new RuntimeException(ex);
        }
    }

}
```

WaitingThread isminde bir *Thread* tanımladık. Bu iş parçacığını bekleteceğiz. Gördüğünüz gibi, iş parçacığı çalıştığı zaman konsola beklediğimizi yazdırıyoruz. Daha sonra *await()* metoduyla bu iş parçacığını bekletiyoruz. İş parçacığı beklemesini bitirdikten sonra konsola bilgilendirme yazısı yazacaktır.

```
CountDownLatch latch = new CountDownLatch(10);
WaitingThread waitingThread = new WaitingThread(latch);
waitingThread.start();

for (int i = 1; i <= 10; i++)
{
    MyThread myThread = new MyThread(latch, String.valueOf(i));
    myThread.start();
}
```

Son olarak yukarıdaki kodu yazıyoruz. Burada öncelikle *CountDownLatch* nesnesi oluşturduk ve parametre olarak 10 değerini verdik. Yani, 10 adet iş parçacığı çalışana kadar bekleyeceğiz. Daha sonra bekleteceğimiz iş parçacığını ve 10 adet bekleyeceğimiz iş parçacığını oluşturduk ve başlattık. Bu kodu çalıştırırsanız aşağıdaki gibi bir çıktı alırsınız:

```
Bekleyen thread bekliyor...
3 çalışıyor...
4 çalışıyor...
1 çalışıyor...
9 çalışıyor...
8 çalışıyor...
7 çalışıyor...
5 çalışıyor...
2 çalışıyor...
10 çalışıyor...
6 çalışıyor...
Bekleyen thread çalıştı.
```

Gördüğünüz gibi, beklettiğimiz iş parçacığı diğer 10 iş parçacığı tamamlanana kadar beklemiştir. Bu kodu defalarca kez çalıştırabilirsiniz; iş parçacıklarının çalışma sırası farklı olabilir, fakat her seferinde beklettiğimiz iş parçacığı en son çalışacaktır.

CyclicBarrier

CyclicBarrier, *CountDownLatch* nesnesine benzer. Belli sayıda iş parçacığını bekletmeye yarar. Farkı şudur: *CountDownLatch* nesnesini başka bir iş parçacığını bekletmek için kullanırız; fakat *CyclicBarrier* çalışan iş parçacıklarını bekletmek için kullanılır. Aşağıdaki örneği inceleyelim:

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class MyThread extends Thread
{

    private final CyclicBarrier barrier;

    public MyThread(CyclicBarrier barrier, String name)
    {
        this.barrier = barrier;
        setName(name);
    }

    @Override
    public void run()
    {
        try
        {
            System.out.println(getName() + " çalışıyor...");
            barrier.await();
            System.out.println(getName() + " devam ediyor...");
        }
        catch (InterruptedException | BrokenBarrierException ex)
        {
            throw new RuntimeException(ex);
        }
    }

}
```

MyThread isminde bir *Thread* tanımladık. Bu iş parçacığı çalışırken konsola çalıştığını yazdıracağız, daha sonra *CyclicBarrier* nesnesinin *await()* metodunu kullanarak bekleteceğiz. Belirlediğimiz sayıda iş parçacığı bu bariyere çarptıktan sonra iş parçacıkları devam edecektir ve konsola devam ettiğini yazacaktır.

Şimdi aşağıdaki kodu çalıştıralım:

```
CyclicBarrier barrier = new CyclicBarrier(10);

for (int i = 1; i <= 10; i++)
{
    MyThread thread = new MyThread(barrier, String.valueOf(i));
    thread.start();
}
```

Bu kodu çalıştırırsanız şuna benzer bir çıktı alırsınız:

```
1 çalışıyor...
3 çalışıyor...
2 çalışıyor...
7 çalışıyor...
5 çalışıyor...
4 çalışıyor...
6 çalışıyor...
10 çalışıyor...
8 çalışıyor...
9 çalışıyor...
2 devam ediyor...
4 devam ediyor...
3 devam ediyor...
5 devam ediyor...
6 devam ediyor...
10 devam ediyor...
7 devam ediyor...
8 devam ediyor...
9 devam ediyor...
1 devam ediyor...
```

Gördüğünüz gibi, 10 adet iş parçacığının hepsi çalışmaya başlamış ve *await()* metoduyla beklemeye geçmiştir. Ne zaman ki onuncu iş parçacığı da bu bariyere çarpmıştır, bütün iş parçacıkları kaldıkları yerden devam etmiştir.

CyclicBarrier oluştururken isterseniz ikinci parametre olarak bir *Runnable* nesnesi verebilirsiniz. Belirttiğiniz sayıda iş parçacığı bariyere çarptıktan sonra bu *Runnable* nesnesi çalıştırılır. Örneğin, yukarıdaki kodu aşağıdaki gibi değiştirelim:

```
CyclicBarrier barrier = new CyclicBarrier(10, new Runnable()
{
    @Override
    public void run()
    {
        System.out.println("Bariyer delindi!");
    }
});

for (int i = 1; i <= 10; i++)
{
    MyThread thread = new MyThread(barrier, String.valueOf(i));
    thread.start();
}
```

Şimdi bu kodu çalıştırırsanız aşağıdaki gibi bir çıktı alırsınız:

```
8 çalışıyor...
10 çalışıyor...
6 çalışıyor...
4 çalışıyor...
3 çalışıyor...
1 çalışıyor...
7 çalışıyor...
9 çalışıyor...
5 çalışıyor...
2 çalışıyor...
Bariyer delindi!
2 devam ediyor...
7 devam ediyor...
3 devam ediyor...
1 devam ediyor...
6 devam ediyor...
5 devam ediyor...
8 devam ediyor...
10 devam ediyor...
9 devam ediyor...
4 devam ediyor...
```

Gördüğünüz gibi, bariyer delindikten sonra, diğer iş parçacıklarından önce parametre olarak verdiğimiz *Runnable* nesnesi çalıştırılmıştır.

Exchanger<T>

Senkronizasyon nesneleri arasında en ilginç olanı *Exchanger* nesnesidir. İki iş parçacığının birbiriyle veri alışverişi yapmasını sağlar. Jenerik bir sınıftır, transferi yapılacak verinin türü parametre olarak verilir.

Exchanger sınıfının yaptığı işi, birbirini kesen iki caddenin kesişim noktasında buluşan iki arabaya benzetebilirsiniz. Bu arabalardan biri kavşak noktasına diğerinden önce gelirse diğer arabayı bekler. *Exchanger* nesnesini kullanarak iki iş parçacığının bir noktada buluşmasını sağlarız; bunu *exchange()* metoduyla yaparız. Bu metodu daha önce çağıran iş parçacığı diğerini beklemek zorundadır. İki iş parçacığı da bu metodu çağırıp veri alışverişini yaptıktan sonra iş parçacıkları çalışmasına kaldığı yerden devam eder.

Exchanger sınıfının *exchange()* metodunun yapısı aşağıdaki gibidir:

T *exchange*(T *val*)

Metoda parametre olarak verdiğiniz değer diğer iş parçacığına aktarılır. Bu metot diğer iş parçacığından gelen değeri döndürür.

Aşağıdaki örneği inceleyelim:

```
import java.util.Random;
import java.util.concurrent.Exchanger;

public class MyThread extends Thread
{

    private final Exchanger<Integer> exchanger;

    public MyThread(Exchanger<Integer> exchanger, String name)
    {
        this.exchanger = exchanger;
        setName(name);
    }

    @Override
    public void run()
    {
        Random random = new Random();
        int value = random.nextInt(50);
        int ms = random.nextInt(5000) + 5000;

        try
        {
            System.out.println(getName() +
                  " şu değeri üretti: " + value);
            System.out.println(getName() + " " + ms
                  + " milisaniye bekliyor...");

            Thread.sleep(ms);

            int newValue = exchanger.exchange(value);

            System.out.println(getName() +
                  " - yeni değer: " + newValue);
        }
        catch (InterruptedException ex)
        {
            System.err.println(getName() + " kesildi!");
        }
    }

}
```

MyThread adında bir *Thread* oluşturduk. Bu iş parçacığı çalışırken önce 0 ile 50 arasında rastgele bir sayı oluşturacak. Daha sonra 5 ile 10 saniye arasında bir değer üretecek ve bu kadar bekleyecek. Son olarak *exchanger* nesnesini

kullanarak ürettiği rastgele değeri diğer iş parçacığına aktaracak ve karşılık olarak ondan gelen değeri alacak.

```
Exchanger<Integer> exchanger = new Exchanger<>();

MyThread a = new MyThread(exchanger, "A");
MyThread b = new MyThread(exchanger, "B");

a.start();
b.start();
```

Yukarıdaki kodu çalıştırırsanız aşağıdaki gibi bir çıktı alırsınız:

```
B şu değeri üretti: 43
A şu değeri üretti: 25
B 9445 milisaniye bekliyor...
A 6597 milisaniye bekliyor...
B - yeni değer: 25
A - yeni değer: 43
```

Gördüğünüz gibi, B iş parçacığı A'dan daha uzun süre uyuduğu için A iş parçacığı B'yi beklemek zorunda kalmıştır.

Executor nesneleri

Çok fazla iş parçacığı çalıştıracağınız uygulamalarda *ExecutorService* kullanmanız daha iyi olur. *Executor* nesneleri iş parçacıklarının çalıştırılması ve kontrolü için alternatif bir yöntem sunar. *ExecutorService* oluşturmak için *Executors* sınıfının statik metotlarını kullanırız. Bu metotların bazılarını inceleyelim:

static ExecutorService newCachedThreadPool()	İhtiyaç oldukça yeni iş parçacığı oluşturan bir **iş parçacığı havuzu** (*thread pool*) oluşturur. Ayrıca, daha önce oluşturulmuş ve müsait durumdaki iş parçacıklarını da kullanır. Kısa ömürlü birçok iş parçacığı çalıştırmanız gereken durumlarda bu havuzu kullanmalısınız.

static ExecutorService newFixedThreadPool(**int** nThreads)	Parametre olarak belirtilen sayıda iş parçacığı oluşturur ve sürekli bunları kullanır. Herhangi bir anda çalışan iş parçacığı sayısı bu değerden yüksek olmaz. Bütün iş parçacıkları çalışırken yeni bir görev eklenirse, bunlardan biri müsait duruma geçene kadar bekletilir.
static ExecutorService newSingleThreadExecutor()	Yalnızca bir adet iş parçacığı kullanan bir havuz oluşturur. Aynı anda sadece bu iş parçacığı çalışır. Dolayısıyla görevler sırasıyla çalıştırılır.
static ScheduledExecutorService newScheduledThreadPool(**int** corePoolSize)	Belirtilen sayıda iş parçacığı içeren bir havuz oluşturur. Bu havuz, görevleri zamanlamak için kullanılır. Belli bir süre sonra veya belli sürelerde tekrar eden işleri çalıştırmak için bu havuz kullanılabilir.
static ScheduledExecutorService newSingleThreadScheduledExecutor()	Yalnızca bir adet iş parçacığı kullanan bir zamanlayıcı havuz oluşturur.

Bir *ExecutorService* oluşturulduktan sonra *submit()* metodunu kullanarak *Runnable* türünde görevler verilir. Bu görev *Executor* nesnesinin yapısına göre gelecekte çalıştırılır:

Future<?> submit(**Runnable** task)

Bu metot *Future* türünde bir nesne döndürür. Bu nesneyi kullanarak parametre olarak verdiğiniz görevi iptal edebilir veya görevin durumunu öğrenebilirsiniz.

Her *ExecutorService* nesnesi kullanımı bittikten sonra kapatılmalıdır. Bunun için iki farklı metodu kullanabilirsiniz:

void shutdown()

Bu metot *Executor* nesnesinin içinde bulunan çalıştırılmamış bütün görevleri çalıştırır ve daha sonra nesneyi sonlandırır. Bu metodu çağırırsanız yeni görev ekleyemezsiniz.

List<Runnable> shutdownNow()

Bu metot *Executor* nesnesini anında kapatır. O anda çalıştırılmakta olan iş parçacığı sonlandırılır. Henüz çalıştırılmamış görev varsa bunları bir liste halinde döndürür.

Şimdi *ExecutorService* kullanan bir örnek yapalım:

```
import java.util.Random;
import java.util.concurrent.CountDownLatch;

public class MyRunnable implements Runnable
{

    private final Random random = new Random();
    private final CountDownLatch latch;
    private final String name;

    public MyRunnable(CountDownLatch latch, String name)
    {
        this.latch = latch;
        this.name = name;
    }

    @Override
    public void run()
    {
        try
        {
            int ms = random.nextInt(3000) + 3000;
            System.out.println(name + " - " + ms
                  + " milisaniye bekliyor...");

            Thread.sleep(ms);

            System.out.println(name + " - tamamlandı.");
        }
        catch (InterruptedException ex)
        {
            System.err.println(name + " kesildi!");
        }
        finally
        {
            latch.countDown();
        }
    }

}
```

MyRunnable adında, *Runnable* arayüzünü uygulayan bir sınıf oluşturduk. Bu sınıfın *run()* metodu çalıştırılınca iş parçacığını 3 ile 6 saniye arasında bekleteceğiz. Daha sonra yapılandırıcıda parametre olarak aldığımız *CountDownLatch* nesnesinin *countDown()* metodunu çağıracağız.

```
final int poolSize = 10;
CountDownLatch latch = new CountDownLatch(poolSize);
ExecutorService executor = Executors.newCachedThreadPool();

for (int i = 1; i <= poolSize; i++)
{
    MyRunnable runnable = new MyRunnable(latch, String.valueOf(i));
    executor.submit(runnable);
}

try
{
    latch.await();
    executor.shutdownNow();
}
catch (InterruptedException ex)
{
    throw new RuntimeException(ex);
}
finally
{
    System.out.println("Bütün thread'ler çalıştı!");
}
```

Yukarıda 10 adet *MyRunnable* nesnesi oluşturuyoruz. Bu nesneleri parametre olarak *executor* nesnesine veriyoruz. Bütün iş parçacıkları çalıştıktan sonra *shutdownNow()* metoduyla *ExecutorService* nesnesini kapatıyoruz.

Bu kodu çalıştırırsanız şuna benzer bir çıktı alırsınız:

```
9 - 4094 milisaniye bekliyor...
2 - 3461 milisaniye bekliyor...
6 - 5514 milisaniye bekliyor...
10 - 5548 milisaniye bekliyor...
5 - 3009 milisaniye bekliyor...
4 - 4641 milisaniye bekliyor...
3 - 3328 milisaniye bekliyor...
1 - 5452 milisaniye bekliyor...
7 - 3079 milisaniye bekliyor...
8 - 5875 milisaniye bekliyor...
5 - tamamlandı.
7 - tamamlandı.
3 - tamamlandı.
2 - tamamlandı.
9 - tamamlandı.
4 - tamamlandı.
1 - tamamlandı.
6 - tamamlandı.
10 - tamamlandı.
8 - tamamlandı.
Bütün thread'ler çalıştı!
```

Son bir not olarak şunu belirtelim: eğer kullanılan bir *ExecutorService* nesnesini kapatmazsanız oluşturduğu iş parçacıkları hafızada kalmaya devam edecektir. Dolayısıyla programınız sonlanmayacaktır.