

The background is a complex digital-themed composition. The left half features a warm color gradient from orange to red, overlaid with a network of white dots and lines, and scattered binary code (0s and 1s). The right half is a darker blue, showing a blurred image of a person's hands typing on a laptop keyboard, with binary code and network patterns also visible.

# JAVA CONCURRENCY

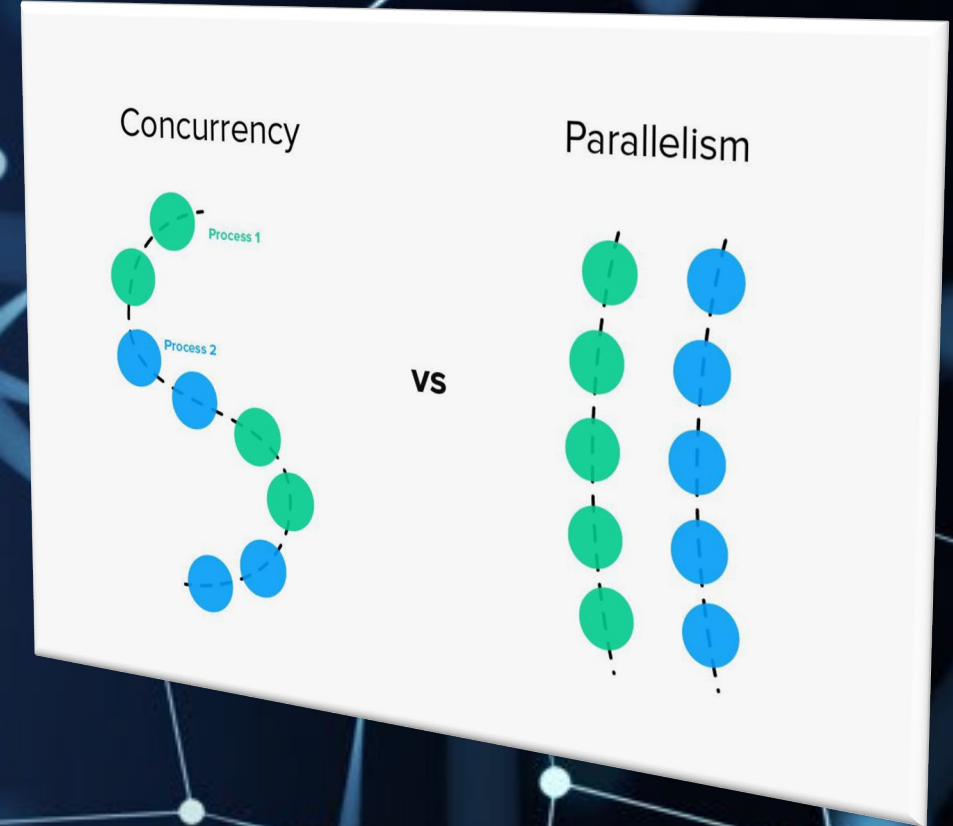
Muhammet Ali KAYA

# Concurrency – Eşzamanlılık

Tüm bilgisayar kullanıcıları, bilgisayarın aynı anda bir çok işlemi yapmasını bekler. Örneğin; müzik dinliyorken kod yazmak. Oyun oynarken bunu canlı yayınla kitlelere ulaştırmak.

Yani bilgisayarlar uzun zamandır bu işi yapabilmekte. Öyleyse bizler neden bilgisayarın bu özelliğini kodlamalarımız üzerinde yapmıyoruz?

Java dili, 5.0 sürümünden bu tarafa «java.util.concurrency» kitaplığı ile bunu desteklemektedir.





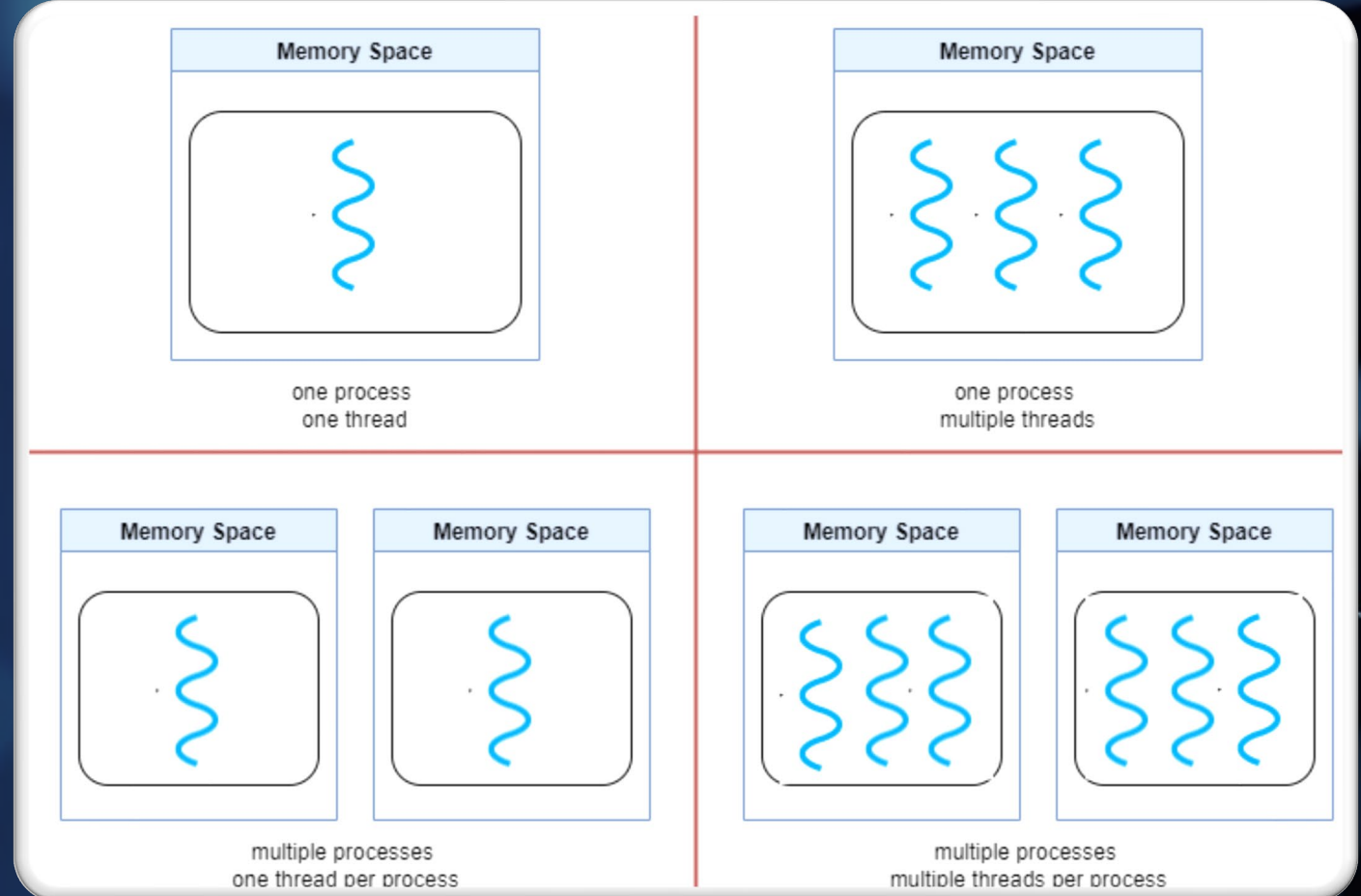
# Kavramlar

**Processes:** bir işlem parçacığı işletim sistemi tarafından programlanabilen en küçük yürütme birimidir. Her işlemin kendi bellek alanı vardır.

Bizim tek işlem gördüğümüz şeyler aslında organize olmuş bir çok işlemin işlemesi ile yürür.

JVM de bir çok java uygulaması tek bir işlem olarak yürütülür. Ancak, ProcessBuilder ile bir nesnenin ilk işlemlerde koşturulması sağlanabilir.

**Thread:** iplik denilen bu yapılar, işlemler içinde bulunur ve bir yürütme ortamını bizim için sağlarlar. Her işlem için de en az bir iplik vardır. Temel olarak kullanılan ipliğe MainThread adı verilir.



# Java Memory Model

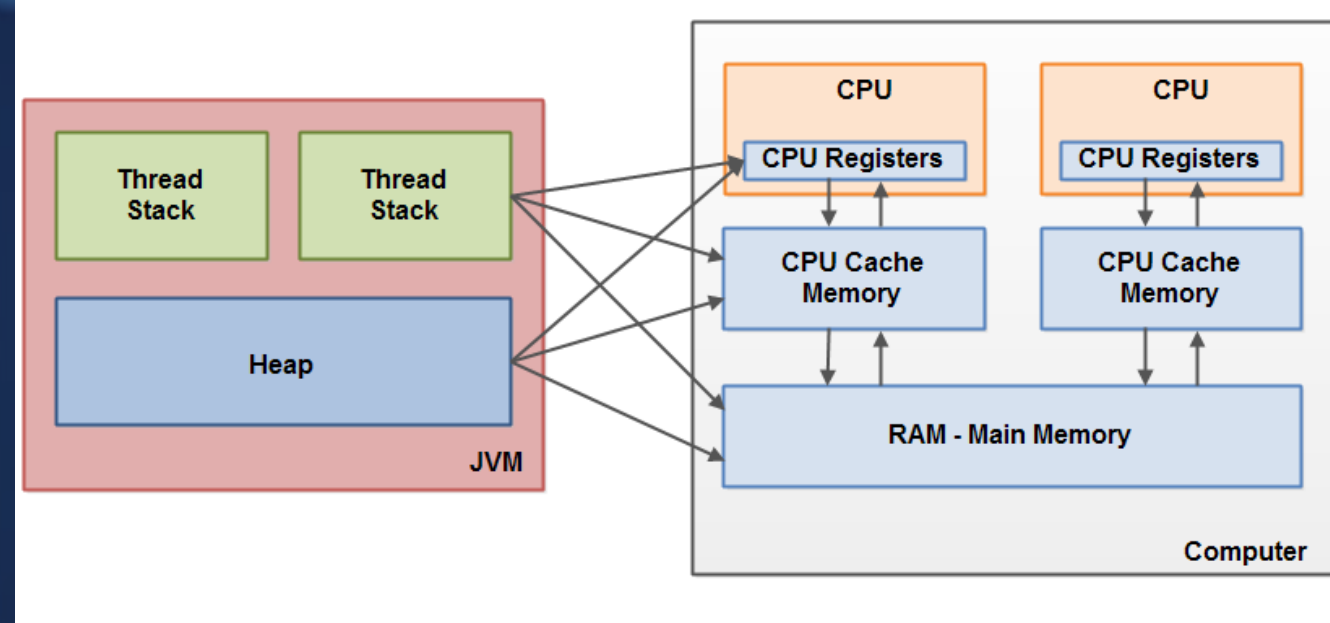
Java 5.0 dan beridir aynı bellek modeli kullanılmaktadır. Doğru ve en azından çalışan uygulamalar yazmak istiyorsanız javanın belleğinizi nasıl yönettiğini anlamalısınız.

Java, farklı iş parçacıklarının diğer iş parçacıkları tarafından paylaşılan değerlere işlenmiş olan bilgilere ne zaman ve nasıl erişim yapılacağını senkronizasyonun nasıl olacağını yönetir.

JVM, kendi belleğini iş parçacıkları arasında böler, çalışmakta olan her bir iş parçacığının kendine ait bir Stack i var dır ve bununla yürüttüğü işlemler hakkında bilgi alırız.

**DİKKAT!!**

İş parçacıklarının(Thread Stack) kendi yığınlarında oluşan değişkenler sadece o iş parçacığı için geçerlidir dışarıdan erişilemezler. Bunlar tam olarak ilkel veri türlerinin kendisidir.



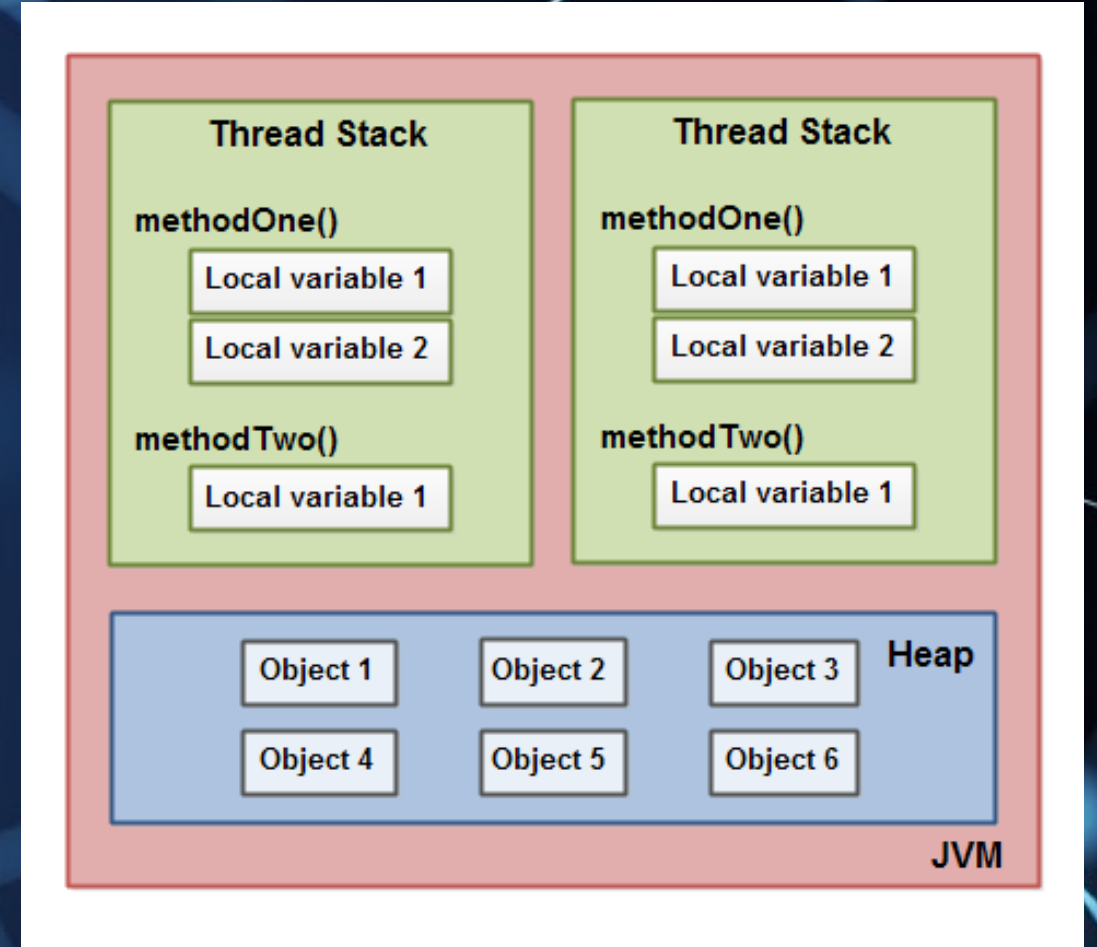
# Java Memory Model

Heap, java uygulamanızda oluşturulan tüm nesneleri içerir. Bir nesnenin oluşturulup yerel bir değişkene atanmış olması ya da bir nesnenin üye değişkeni olarak oluşturulmuş olması da farketmez heap alanında depolanır.

Yerel bir değişken bir nesneye referans olabilir, ancak bu değişkenin adresi iş parçacığı stack in de depolanırken nesnenin kendisi heap alanında dır.

Bir nesne düşünün içerisinde primitive, references data type lar olsun bu nesnenin kensini de içerisinde konumlanan değişkenleri de heap te depolanır.

static olarak tanımladığımız değişkenlerde sınıf tanımı ile birlikte heap te konumlanır.



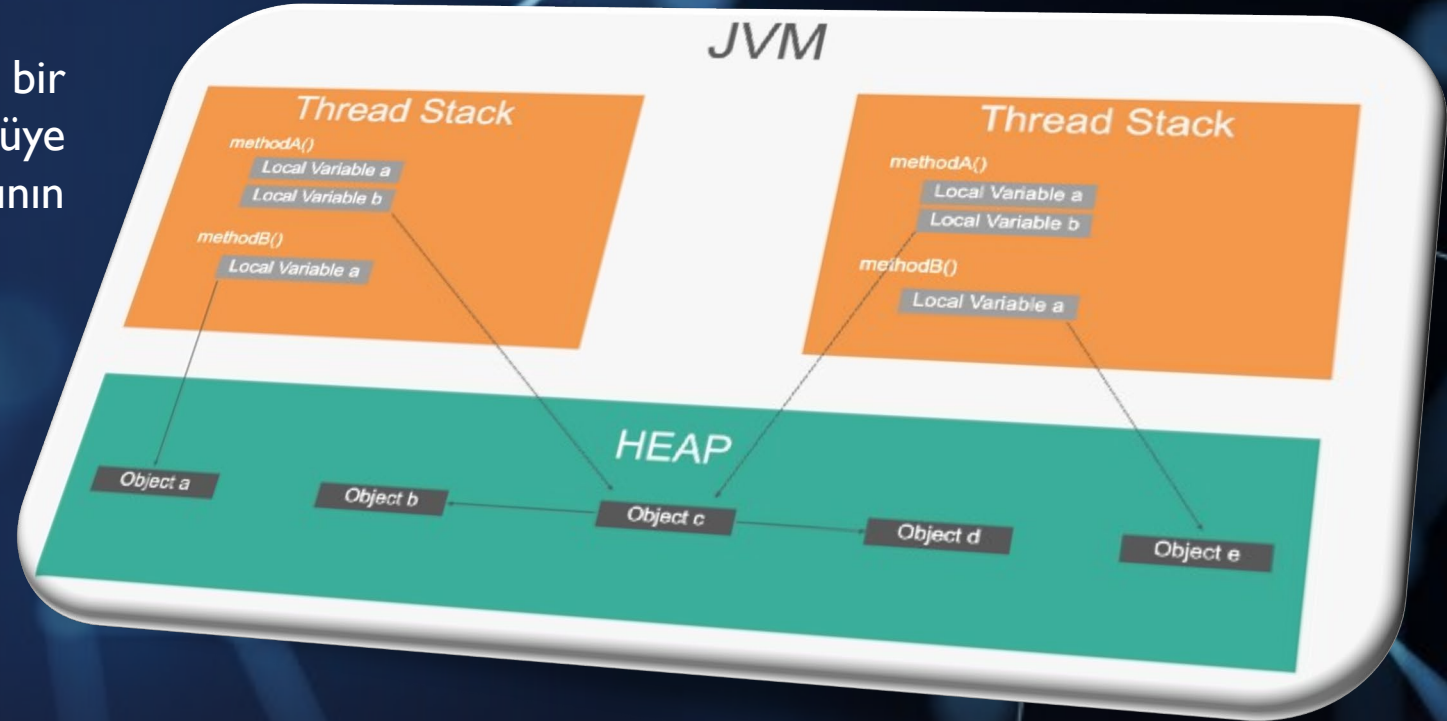


# Java Memory Model

Heap üzerindeki nesnelere, nesneye referansı olan tüm iş parçacıkları tarafından erişilebilir. Bir iş parçacığının bir nesneye erişimi olduğunda, o nesnenin üye değişkenlerine de erişebilir.

**DİKKAT!!!**

İki iş parçacığı aynı anda aynı nesne üzerinde bir yöntemi çağırırsa, her ikisinin de nesnenin üye değişkenlerine erişimi olur, ancak her iş parçacığının yerel değişkenlerin kendi kopyası olur

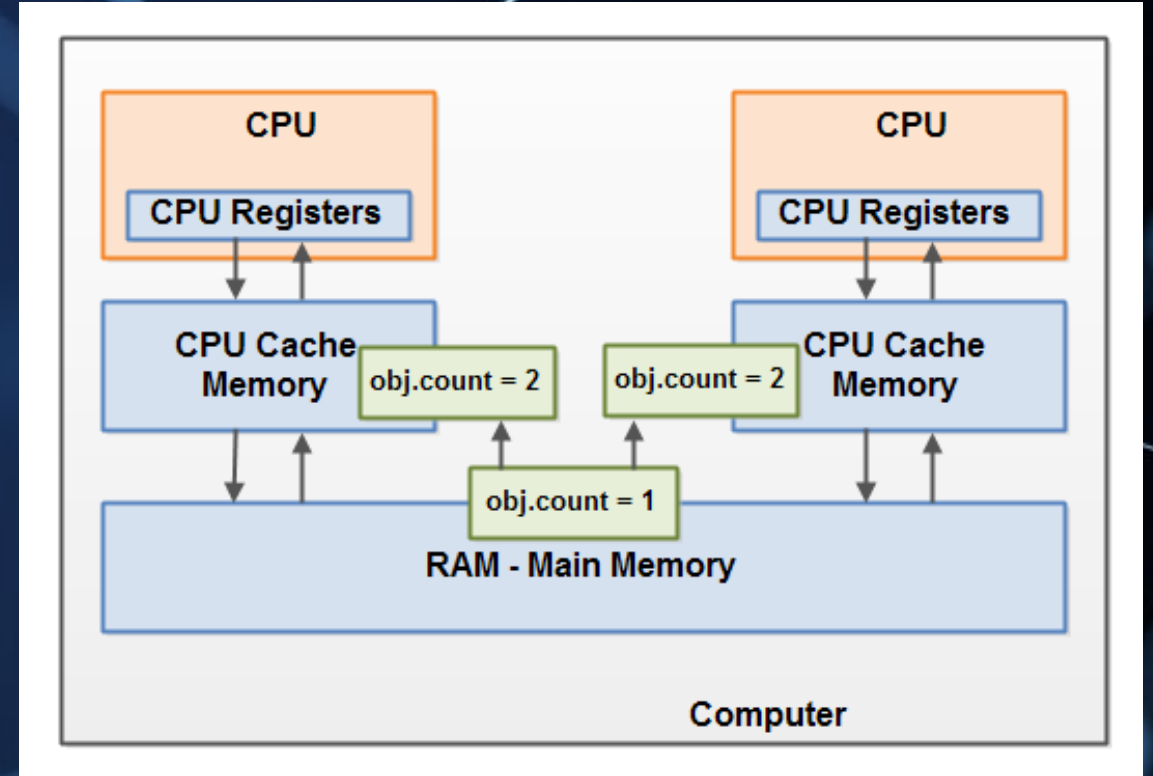


# Java Memory Model

İki veya daha fazla iş parçacığı bir nesneyi paylaşıyorsa, bildirimlerin veya senkronizasyonun doğru kullanımı olmadan, bir iş parçacığı tarafından paylaşılan nesnede yapılan güncellemeler diğer iş parçacıkları tarafından görülmeyebilir.

**!! SORUN !!**

Paylaşılan nesnenin başlangıçta ana bellekte depolandığını hayal edin. CPU'da çalışan bir iş parçacığı daha sonra paylaşılan nesneyi CPU önbelleğine okur. Orada paylaşılan nesnede bir değişiklik yapar. CPU önbelleği ana belleğe geri yüklenmediği sürece, paylaşılan nesnenin değişen sürümü diğer CPU'larda çalışan iş parçacıkları tarafından görülmez. Bu şekilde, her iş parçacığı, paylaşılan nesnenin kendi kopyasıyla sonuçlanabilir, her kopya farklı bir CPU önbelleğinde bulunur



# Java Memory Model

!! ÇÖZÜM !!

Bu sorunu çözmek için Java'nın volatile anahtar sözcüğünü kullanabilirsiniz. Anahtar volatile sözcük, belirli bir değişkenin doğrudan ana bellekten okunmasını ve güncellendiğinde her zaman ana belleğe yazılmasını sağlayabilir.



# Java Memory Model

Uygulamalarınız ile ilgili Bellek kullanımı ve detaylarını merak ediyor ve incelemek istiyorsanız.

Link:

<https://docs.oracle.com/javase/8/docs/technotes/guide/s/troubleshoot/tooldescr007.html>

Uygulamanızın KM de ne kadar yaktığını öğrenmek istiyorsanız.

```
C:\> java -XX:NativeMemoryTracking=summary -Xms300m -Xmx300m -XX:+UseG1GC -jar app.jar
```

```
Total: reserved=664192KB, committed=253120KB <--- total memory tracked by Native Memory Tracking

-      Java Heap (reserved=516096KB, committed=204800KB) <--- Java Heap
      (mmap: reserved=516096KB, committed=204800KB)

-      Class (reserved=6568KB, committed=4140KB) <--- class metadata
      (classes #665) <--- number of loaded classes
      (malloc=424KB, #1000) <--- malloc'd memory, #number of malloc
      (mmap: reserved=6144KB, committed=3716KB)

-      Thread (reserved=6868KB, committed=6868KB)
      (thread #15) <--- number of threads
      (stack: reserved=6780KB, committed=6780KB) <--- memory used by thread stacks
      (malloc=27KB, #66)
      (arena=61KB, #30) <--- resource and handle areas

-      Code (reserved=102414KB, committed=6314KB)
      (malloc=2574KB, #74316)
      (mmap: reserved=99840KB, committed=3740KB)

-      GC (reserved=26154KB, committed=24938KB)
      (malloc=486KB, #110)
      (mmap: reserved=25668KB, committed=24452KB)

-      Compiler (reserved=106KB, committed=106KB)
      (malloc=7KB, #90)
      (arena=99KB, #3)

-      Internal (reserved=586KB, committed=554KB)
      (malloc=554KB, #1677)
      (mmap: reserved=32KB, committed=0KB)

-      Symbol (reserved=906KB, committed=906KB)
      (malloc=514KB, #2736)
      (arena=392KB, #1)

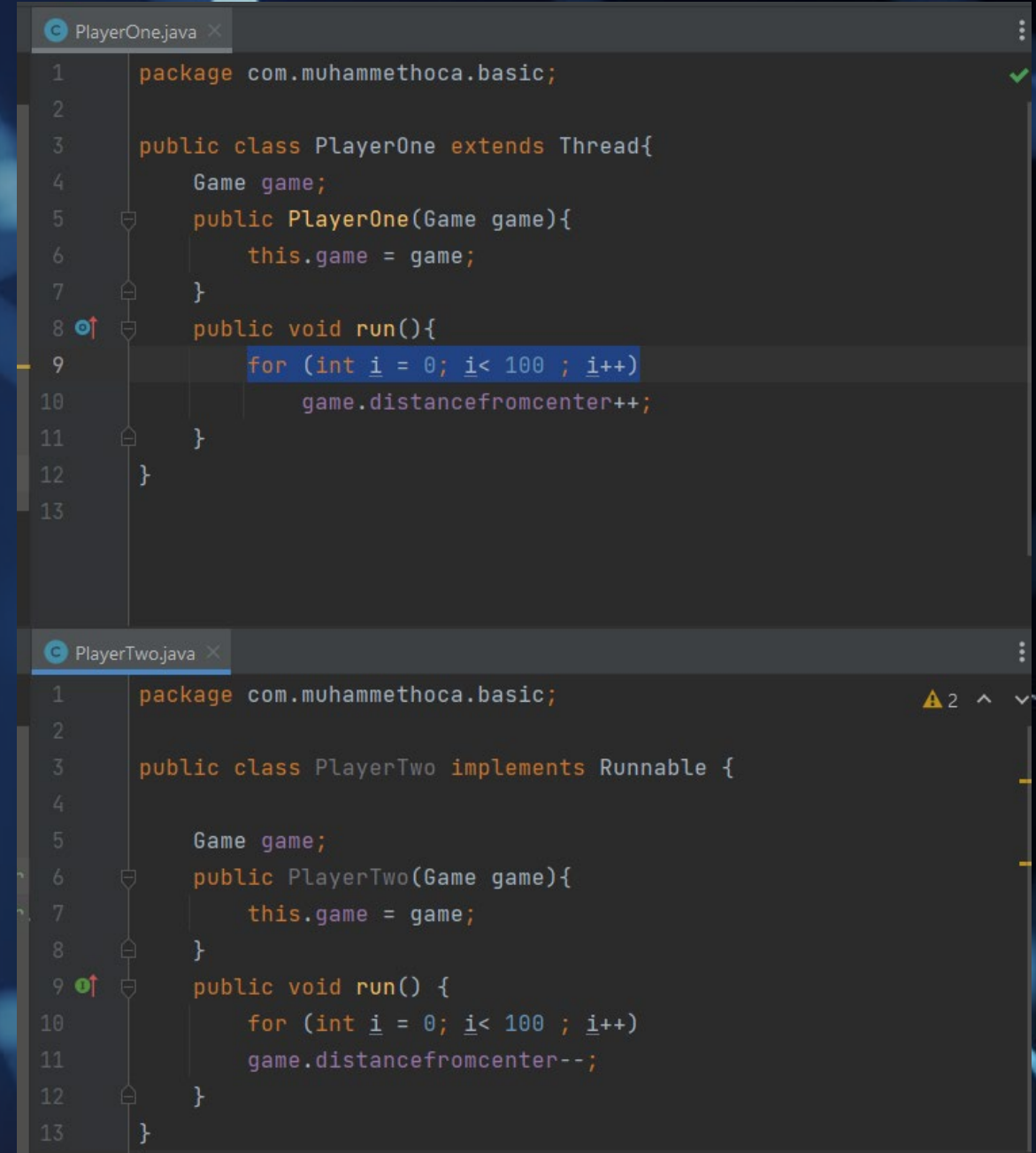
-      Memory Tracking (reserved=3184KB, committed=3184KB)
      (malloc=3184KB, #300)
```

# Nasıl Kullanılır?

Bir Thread oluşturmanın temelde iki yöntemi vardır;

- Bunlardan ilki Runnable interface ini kullanarak oluşturabiliriz.
- İkinci olarakta Thread sınıfını miras alarak bir thread oluşturma yöntemine başvurabiliriz.

Bunlar dışında farklı şekillerde de threadler oluşturabiliriz.



```
PlayerOne.java
1 package com.muhammethoca.basic;
2
3 public class PlayerOne extends Thread{
4     Game game;
5     public PlayerOne(Game game){
6         this.game = game;
7     }
8     public void run(){
9         for (int i = 0; i < 100 ; i++)
10             game.distancefromcenter++;
11     }
12 }
13
```

```
PlayerTwo.java
1 package com.muhammethoca.basic;
2
3 public class PlayerTwo implements Runnable {
4
5     Game game;
6     public PlayerTwo(Game game){
7         this.game = game;
8     }
9     public void run() {
10         for (int i = 0; i < 100 ; i++)
11             game.distancefromcenter--;
12     }
13 }
```

# Nasıl Kullanılır?

Kullanım şekillerinde, Lambda Expressions ları kullanarak işlem yapabilirsiniz.

```
public static void main(String[] args) {  
    /**  
     * Using 1. Lambda Expression  
     */  
    new Thread( () -> System.out.println("Lambda 1 Hello World")).start();  
    /**  
     * Using 2. Lambda Expression  
     */  
    new Thread( () -> {  
        for (int i = 0; i < 1; i++) {  
            System.out.println("Lambda 2 Hello World");  
        }  
    }).start();  
    /**  
     * Using 3. Lambda Expression  
     */  
    Runnable runnable = () -> System.out.println("Lambda 3 Hello World");  
    /**  
     * Using 4. Lambda Expression  
     */  
    Runnable runnable2 = () -> {  
        for (int i = 0; i < 4; i++) {  
            System.out.println("Lambda 4 Hello World...: " + i);  
        }  
    };  
    System.out.println("Begin");  
    runnable.run();  
    runnable2.run();  
    runnable.run();  
    System.out.println("End");  
}
```



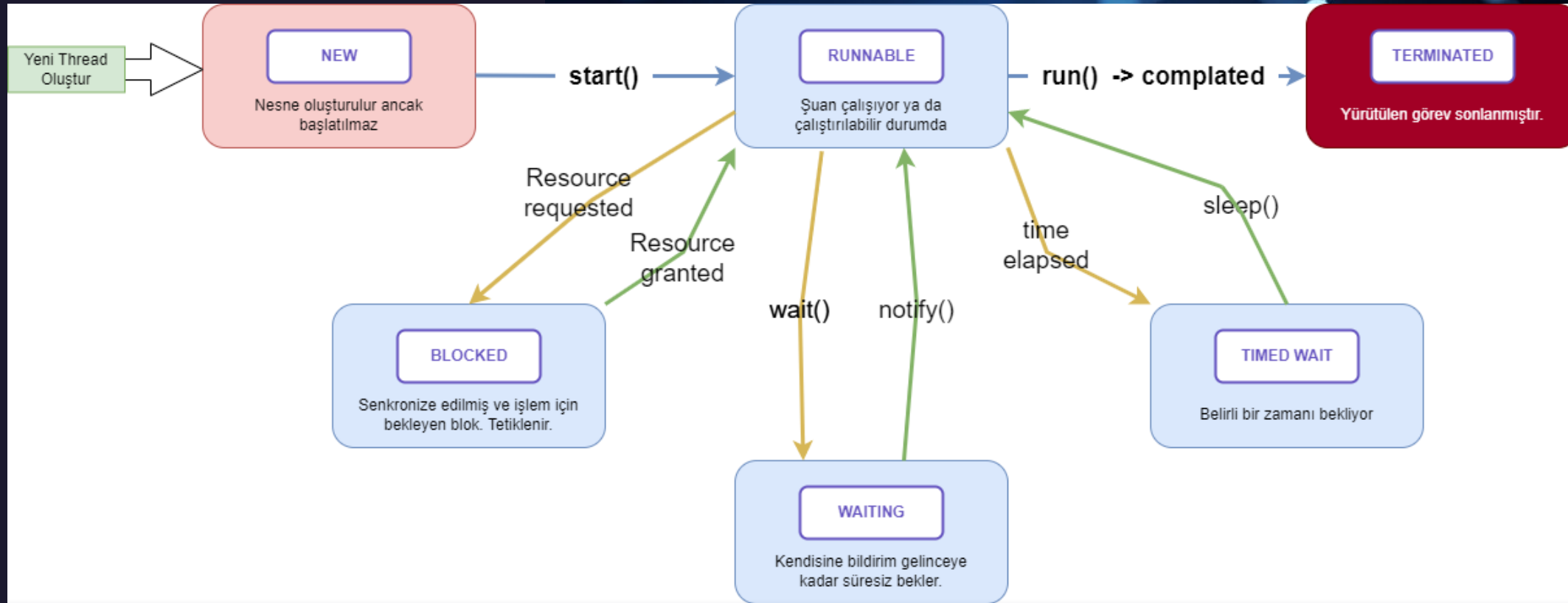
# Hangisini Kullanmalıyım?

- Thread, miras alınarak kullanılabilir ancak, bir sınıfın tek miras hakkının bununla harcanması pek mantıklı değildir. Uygulama çatısında miras verilecek ana sınıflar olabilir. Bu nedenle thread kullanmak mantıklı değildir.
- Runnable, java da bir den fazla implemets yapılabilir. Java da olmamasına rağmen bu yapı kullanılarak multiple inheritance sağlanabilir. Runnable olarak implement edilen bir sınıf thread değildir ve başlatılamaz. Bu nedenle, istenildiği kadar thread içinde çalıştırılabilir.

## THREAD RUNNABLE



# Threadlerin Yaşam Döngüsü



# Pooling with Sleep

- Çok iş parçacıklı programlama aynı anda birden çok görevi yürütmenize izin verir, ancak bir iş parçacığının genellikle başka bir iş parçacığının sonuçlarını beklemesi gerekebilir. Bu gibi problemleri çözmenin birden çok yöntemi vardır bunlardan ilki `Thread.sleep([time])` kullanmaktır.

```
public class Runner_LifeCycle_Sleep {  
    private static long counter = 0;  
    public static void main(String[] a) {  
        new Thread(() ->  
        {  
            for(long i = 0; i < 10_000_000_000L; i++) counter++;  
        }).start();  
        while(counter < 10_000_000_000L) {  
            System.out.println("Not reached yet");  
            try {  
                Thread.sleep(1_000); // 1 SECOND  
            } catch (InterruptedException e) {  
                System.out.println("Interrupted!");  
            }  
        }  
        System.out.println("Reached: "+counter);  
    }  
}
```

```
IDEA 2021.3.2\bin" -Dfile.encoding=UTF-8  
Not reached yet  
Not reached yet  
Not reached yet  
Not reached yet  
Not reached yet  
Reached: 10000000000
```



# Interrupting a Thread

Bir önce ki çözümümüzde çalışan Thread in işini bitirmesi için bekleme süresini kendimiz belirledik ve buna göre işlem yaptık.

Ancak, şöyle bir durum var; biz Thread in işini bitirmesi için bekliyoruz, bekleme süremiz 1 saniye olsun fakat thread işini 0.5 saniye de bitiriyor diyelim, böyle bir durumda yapmış olduğumuz bekleme işlemi bize pahalıya patlayacaktır. Neyse ki bunun çözümü var. İlgili thread in bekleme işlemini kesmek ve kodun devam etmesini sağlamak.

```
public class Runner_LifeCycle_Interrupting {  
    private static int counter = 0;  
    public static void main(String[] a) {  
        final var mainThread :Thread = Thread.currentThread();  
        new Thread(() ->  
        {  
            for (int i = 0; i < 1_000_000; i++) counter++;  
            mainThread.interrupt();  
        }).start();  
        while (counter < 1_000_000) {  
            System.out.println("Not reached yet");  
            try {  
                Thread.sleep(1_000); // 1 SECOND  
            } catch (InterruptedException e) {  
                System.out.println("Interrupted!");  
            }  
        }  
        System.out.println("Reached: " + counter);  
    }  
}
```

```
.Runner_LifeCycle_Interrupting  
Not reached yet  
Interrupted!  
Reached: 1000000  
  
Process finished with exit code 0
```

# Single Executor

Bir ya da daha çok Thread farketmez bu işlemleri yönetmek ve kodlamak karmaşık olabilir, yaptığınız işe bağlı olarak ta karmaşıklık sizi çileden çıkartabilir. Bu nedenle, iş parçacığı havuzunun yönetimi ve zamanlaması gibi işlemleri sizin adınıza yönetmek ve elinizi güçlendirmek adına Concurrency Api yi kullanmak uygun olacaktır.

ExecutorService size bu yönetimi sunun araçlardan biridir.

**DİKKAT!!!!!!**

Burada, shutdown() methodunu çağırmak önemlidir. Çünkü Executor iş parçacıklarını yürütürken işlerini bitirdikçe diğer iş parçacıklarına izin verir. Eğer bir sonlandırıcı çağırılmaz ise arkaplanda uygulamanız açık kalır ve diğer işlere isTerminated false döndüğü için RejectedExecutionException hatasını fırlatır.

```
public class Runner_ExecutorService {  
  
    public static void main(String[] args) {  
        Runnable gorev = () -> System.out.println("Görev çalıştı");  
  
        Runnable gorev2 = () -> {  
            for (int i = 0; i < 3; i++) {  
                System.out.println("Görev 2 çalıştı");  
            }  
        };  
        ExecutorService executorService = Executors.newSingleThreadExecutor();  
        try {  
            System.out.println("Başladı");  
            executorService.execute(gorev);  
            executorService.execute(gorev2);  
            executorService.execute(gorev);  
            System.out.println("Bitti");  
        } finally {  
            executorService.shutdown();  
        }  
    }  
}
```



## Exceptions

```
public class Runner_Send {  
    public static long whatisloop = 0;  
    public static void main(String[] args) {  
        ExecutorService executorService = Executors.newSingleThreadExecutor();  
        try{  
            Future<?> future = executorService.submit(()->{  
                for(long i=0;i<20_000_000_000l;i++){  
                    whatisloop++;  
                }  
                System.out.println("whatisloop = " + whatisloop);  
            });  
            future.get( timeout: 1, TimeUnit.SECONDS);  
            System.out.println("Reached!");  
        }catch (Exception e) {  
            System.out.println("Not reached in time");  
        }finally{  
            executorService.shutdown();  
        }  
    }  
}
```

Not reached in time  
whatisloop = 20000000000  
  
Process finished with exit code 0

```
public class Runner_Send {  
    public static long whatisloop = 0;  
    public static void main(String[] args) {  
        ExecutorService executorService = Executors.newSingleThreadExecutor();  
        try{  
            Future<?> future = executorService.submit(()->{  
                for(long i=0;i<2_000_000_000l;i++){  
                    whatisloop++;  
                }  
                System.out.println("whatisloop = " + whatisloop);  
            });  
            future.get( timeout: 10, TimeUnit.SECONDS);  
            System.out.println("Reached!");  
        }catch (Exception e) {  
            System.out.println("Not reached in time");  
        }finally{  
            executorService.shutdown();  
        }  
    }  
}
```

whatisloop = 2000000000  
Reached!  
  
Process finished with exit code 0

# Waiting For Results

- Buraya kadar yaptığımız işlemlerde threadler kendi başına işini yürütür ve biter şeklinde idi. Ancak, biz bu görevlerin ne yaptığı hakkında bilgi sahibi değildik, aslında execute() metodu void olduğu için bir bilimiz yoktu. Bu kısmın eksikliğini gidermek için java geliştiricileri send() işlevini eklediler. Bu method bize Future nesnesi döndürmektedir.
- future çalışmakta olan thread için bir bekleme zamanı belirler ve bu zaman diliminde cevap alamaz görev tamamlanamaz ise, concurrentTimeoutException fırlatır.



```

public static void main(String[] args) throws ExecutionException, InterruptedException {
    var service : ExecutorService = Executors.newSingleThreadExecutor();
    try {
        Future<String> result = service.submit(() -> "Merhaba Dünyalı, Ben Java gezegeninden geliyorum.");
        System.out.println(result.get());
    } finally {
        service.shutdown();
    }
}

```

Type parameters: <V> – the result type of method call

@FunctionalInterface

```

public interface Callable<V> {
    V call() throws Exception;
}

```

Computes a result, or throws an exception if unable to do so.  
Returns: computed result  
Throws: Exception – if unable to compute a result

# Introducing Callable

Functional bir interface olan callable sınıfı runnable sınıfına banzer, birkaç küçük farklılığı vardır. Yukarıda yapısını görebiliriz. Callable interface i tanımlandıktan sonra Runnable sınıfına göre daha fazla detay döndüğü için genellikle öncelikli tercih edilir.

Yazımı ve anlaşılması daha temiz olduğu için; kesmeler, thread(ip) ler kullanılarak yazılabilecek kodların yerine kullanılırlar.

```

public class Runner_Scheduling {
    public static void main(String[] args) {
        ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();
        Runnable task1 = () -> System.out.println("Hello Zoo");
        Callable<String> task2 = () -> "Monkey";
        ScheduledFuture<?> r1 = service.schedule(task1, delay: 8, TimeUnit.SECONDS);
        ScheduledFuture<?> r2 = service.schedule(task2, delay: 4, TimeUnit.SECONDS);
        /**
         * DİKKAT!!!
         * Gerçek ortamda planlanan tasklar görev almaya bilir böyle bir durumda
         * iş parçacığı kuyrukta bekler.
         * Ayrıca, ExecutorService zamanından önce kapatılırsa, tüm görevler
         * iptal olur.
         */
    }
}

```

# Scheduling Tasks

Özellikle benim şuan üzerinde çalıştığım uygulamada gün sonunda yapılması gereken o kadar çok görev var ki, hepsini elle yapmak kendime işkence etmek gibi bir şey. Mesela, uygulamada ödül kazananlara bildirim atmak, geçersiz hesaplarla kendine puan verenleri ayıklamak, günlük istatistiklerin ortaklara mail atılması v.s.

Kısaca zamanlanmış, planlanmış işiniz varsa bu tam size göre.

`service.scheduleWithFixedDelay(task1, 0, 2, TimeUnit.MINUTES);` çok kullanışlı olabilir. Zira ne zamana tamamlanacağı belli olmayan fakat biri bittiğinde diğeri belli bir zaman diliminde başlaması gereken görevlerde işe yarayacaktır.

# Increasing Concurrency with Pools

- Çok parçalıklı iş sisteminde, görevleri havuzda bulunan iş parçasığı yürütücüler ile eş zamanlı olarak işleten sistemdir. Daha öncesinde gördüğümüz tüm nesneleri geri döndüren metodlara sahiptir. Burada havuzdan kasıt bir iş parçasığı kümesinin olması ve bu kümenin aynı zamanda bir den fazla işi yürütebiliyor olmasıdır.
- İşler burada ilginç bir hal almaya başlamaktadır. Çünkü farklı iş blokları aynı bellek alanını tükettikleri için beklenmedik durumlar yaşamamak olanaksızdır. Verilerin tutarlılığı kaybolabilir.
- Yanda gördüğünüz şey aynı bellek alanına ulaşan yapının döndüğü cevaptır. Burada aynı rakamı iki defa görebiliriz.

```
public class Runner_ThredSafety {  
    private int The_number_of_participants = 0;  
    private void incrementAndReport() {  
        System.out.print(++The_number_of_participants+" ");  
    }  
    public static void main(String[] args) {  
        ExecutorService service = Executors.newFixedThreadPool( nThreads: 20);  
        try {  
            Runner_ThredSafety manager = new Runner_ThredSafety();  
            for(int i = 0; i < 10; i++)  
                service.submit(() ->  
                    manager.incrementAndReport());  
        } finally {
```

```
C:\Users\MuhammetAli\.jdk\openjdk-11.0.10\bin  
IDEA 2021.3.2\bin" -Dfile.encoding=UTF-8  
1 7 4 6 3 9 8 2 1 5  
Process finished with exit code 0
```

```
C:\Users\MuhammetAli\.jdk\openjdk-11.0.10\bin  
IDEA 2021.3.2\bin" -Dfile.encoding=UTF-8  
9 10 2 4 1 8 6 7 3 5  
Process finished with exit code 0
```



# Increasing Concurrency with Pools

Verilerin güvenliğini sağlamak için, volatile anahtar kelimesini kullanırız. Çünkü geçersiz erişim nedeniyle beklenmeyen bir değer döner. Volatile özelliği, aynı anda sadece bir iş parçasığının bir değişkeni değiştirmesini ve birden çok iş parçasığı arasında okunan verilerin tutarlı olmasını sağlar.

```
public class Runner_ThredSafety {  
    private volatile int The_number_of_participants = 0;  
    private void incrementAndReport() {  
        System.out.print(++The_number_of_participants+" ");  
    }  
    public static void main(String[] args) {  
        ExecutorService service = Executors.newFixedThreadPool( nThreads: 20);  
        try {  
            Runner_ThredSafety manager = new Runner_ThredSafety();  
            for(int i = 0; i < 10; i++)  
                service.submit(() ->  
                    manager.incrementAndReport());  
        } finally {  
            service.shutdown();  
        }  
    }  
}
```

```
IDEA 2021.3.2 (UI) - 011010001  
9 7 1 2 10 3 4 5 8 6  
Process finished with exit code 0
```

# Atomic Class

Atomik, başka bir iş parçasından herhangi bir müdahale olmaksızın tek bir yürütme birimi olarak gerçekleştirilecek bir işlemin özelliğidir.

```
public class Runner_ThredSafety {
    AtomicInteger The_number_of_participants = new AtomicInteger( initialValue: 0);
    //The_number_of_participants.incrementAndGet(); ++value
    //The_number_of_participants.decrementAndGet(); --value
    //The_number_of_participants.get();
    //The_number_of_participants.set(0);
    //The_number_of_participants.getAndIncrement(); value++
    //The_number_of_participants.getAndDecrement(); value--
    private void incrementAndReport() {
        System.out.print((The_number_of_participants.incrementAndGet())+" ");
    }
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool( nThreads: 20);
        try {
            Runner_ThredSafety manager = new Runner_ThredSafety();
            for(int i = 0; i < 10; i++)
                service.submit(() ->
                    manager.incrementAndReport());
        } finally {
            service.shutdown();
        }
    }
}
```

# Synchronized Blocks

Atom sınıfları tek bir değişkeni korumada harika olsa da, bir dizi komutu yürütmeniz veya bir yöntemi çağırmanız gerektiğinde özellikle yararlı değildirler. Örneğin, aynı anda iki atomik değişkeni güncellemek için bunları kullanamayız.

En yaygın teknik, erişimi senkronize etmektir. Kilit olarak da adlandırılır.

Senkron yapılar, aynı belleğe erişimi olan görevlerin aynı anda erişimini kesmek için işe yararlar. Eğer erişim sağlanan nesne kilitli değil ise ona erişim sağlar ve bloklar, diğer görevler kilitli nesneye erişim sağlamazlar. İlgili görev sonlanınca kilidi kaldırır ve diğer görevler sırayla işlemeye devam eder.

```
public class Runner_Synchronized {  
    private int count = 0;  
    private void increment() {  
        synchronized (this) {  
            System.out.print(++count+" ");  
        }  
    }  
    public static void main(String[] args) {  
        ExecutorService service = Executors.newFixedThreadPool( nThreads: 20);  
        try{  
            var manager = new Runner_Synchronized();  
            for (int i = 0; i < 10; i++) {  
                service.submit(manager::increment);  
            }  
        }finally {  
            service.shutdown();  
        }  
    }  
}
```



# Using Concurrent Collections

Concurrent API ile Collection ları rahatlıkla kullanabiliriz. Burada dikkat etmemiz gereken ilk şey, bellek tutarlılık hatalarını ortadan kaldırmaktır.

İki iş parçacığının aynı verinin ne olması gerektiğine dair tutarsızlık yaşadığında sorun oluşur. Burada asıl amacımız bir iş parçacığı yazma işlemi gerçekleştiriyorken diğerinin bu yazma işleminden haberdar olmasını sağlamaktır.

Concurrent API, bünyesinde barındırdığı eş zamanlı sınıflar ile bir den çok iş parçacığının aynı koleksiyona nesne ekleyip çıkardığı anda oluşan yaygın hatalardan kurtulmamızı sağlar.

Ayrıca, Herhangi bir durumda tüm iş parçacıkları koleksiyonun yapısının aynı tutarlı görünüme sahip olmasını sağlar.

Yazdığınız koda güveniyor olsanız da, çok iş parçacıklı işlemlerde istisna fırlatması kaçınılmaz olabilir, ayrıca daha kötüsü datalarınızı farkına varmadan bozabilirsiniz.

Concurrent Koleksiyonları kullanırken, tavsiye edilen tanımlama yöntemi, interface kullanarak tanımlama şeklindedir.

Tabiki çoklu işlemlerden bahsediyoruz, koleksiyonlar içinse senkronize bileşenler olmazsa olmazımızdır. 😊

- synchronizedCollection(Collection<T> c)
- synchronizedList(List<T> list) gibi.

```
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CopyOnWriteArrayList;

public class Runner_ConcurrentCollections {
    public static void main(String[] args) {
        Map<Integer, String> map = new ConcurrentHashMap<>();
        List<String> listem = new CopyOnWriteArrayList<>(List.of("Bugün", "Yarın"));
        map.put(1, "Bugün");
    }
}
```

# Ne dertliymiş bu iş 😊

Kısmen çoklu iş parçacıklarını işledik. Peki temel olarak yaşadığımız sorun nedir? Aslında iki ve daha fazla iş parçacığının beklenmeyen ve istenmeyen şekilde etkileşmesi nedeniyle oluşan sorundur. Örneğin, yemeği yediniz hesap geldi. İki kişi aynı anda adisyonu tuttu işler orada karıştı...

Kilitlenme, her biri diğerini bekleyen iki veya daha fazla iş parçacığı sonsuza kadar engellendiğinde oluşur.





Teşekkürler...