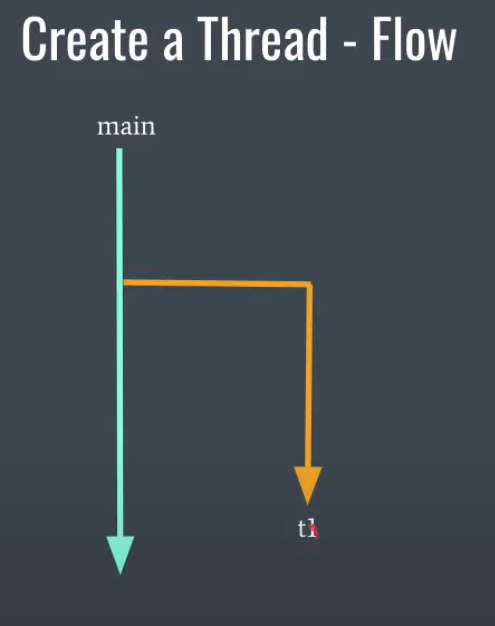
**Java Executor Service Internals**



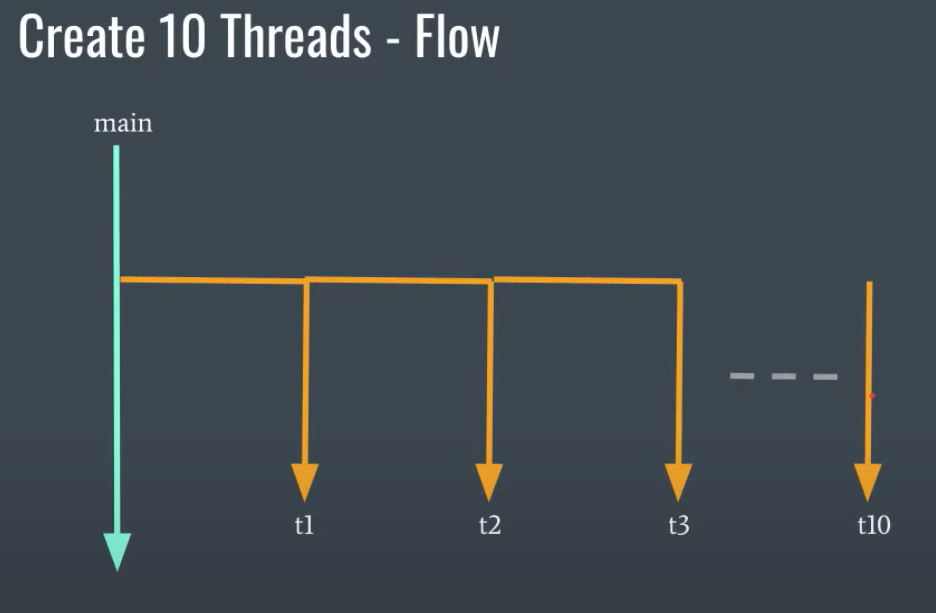
Yukarıda main icinde yeni bir thread acmazsak flow sarı cizginin olmadığı şekildedir.

Ama main thread icinde yeni bir thread olusturursak t1 threadi gibi flow asagıdaki gibidir. Yani main thread icinde farklı threadler acıyoruz.





Yukarıda for icinde 10 tane thread olusturuldu.

Bunun flowu da asagıdaki gibi olur main thread calısırken onun icinde acılan farklı 10 tane thread dallandı. 



Burada 1000 thread olusturmak bu sekilde yönetememe sorunu olusturur.

Multithreading birden fazla thread yarattığımız zaman onun ne zaman calısacağını biz belirleyemiyoruz. Çünkü java JVM üzerinde calısıyor, JVM’de bir process, process altındaki threadleri JVM’in kendisi yönetiyor. JVM’in kendisi operation system isletim sistemi process’idir. İcinde yaratılan threadler de o isletim sisteminin threadidir.

Burada 1000 tane thread olustursak bile bunlar hemen paralelde calısmıyor. Dolayısıyla burada söyle bir durum var. Diyelim bir core var ve saniyede 10 thread’I tamamlayabiliyor. O zaman bu durumda geri kalan 990 tane thread ise calıstıramamaya rağmen bunu olusturuyoruz ve bunların da JVM icinde bir maliyeti var. Cünkü bir threadi yaratıyoruz o bir yer isgal ediyor, referansları var, kendi stackleri var, JVM’ de verdiğimiz parametreler büyükse thread yarattığımız da özel bir alan vs acılacak. Eğer threadi bu kadar kısa zamanda işleyemeyeceksek yani tamamlanayamacaksa bunları önceden yaratmanın anlamı yok.

Günlük hayatta bir bina var bir binanın bir katı, bir katı temizlik olsun, temizlik yaparken burayı hızlı bitirmek icin 100 tane temizlik iscisi koyalım ama o kadar malzeme yok icinde 5 tane temizlik malzemesi var, 100 kisi iceriye koyuyoruz 5 tanesi temizlik yaparken geri kalan 95 tanesi ayak bağı oluyor, yani calısan bir tanesini calısmayan 95 thread yani calısan engel oluyor. Sonucta orada o odada temizlik yaparken calısmayan thread bir yük oluyor

Threadler çok uzun zaman çalışmayan threadler.

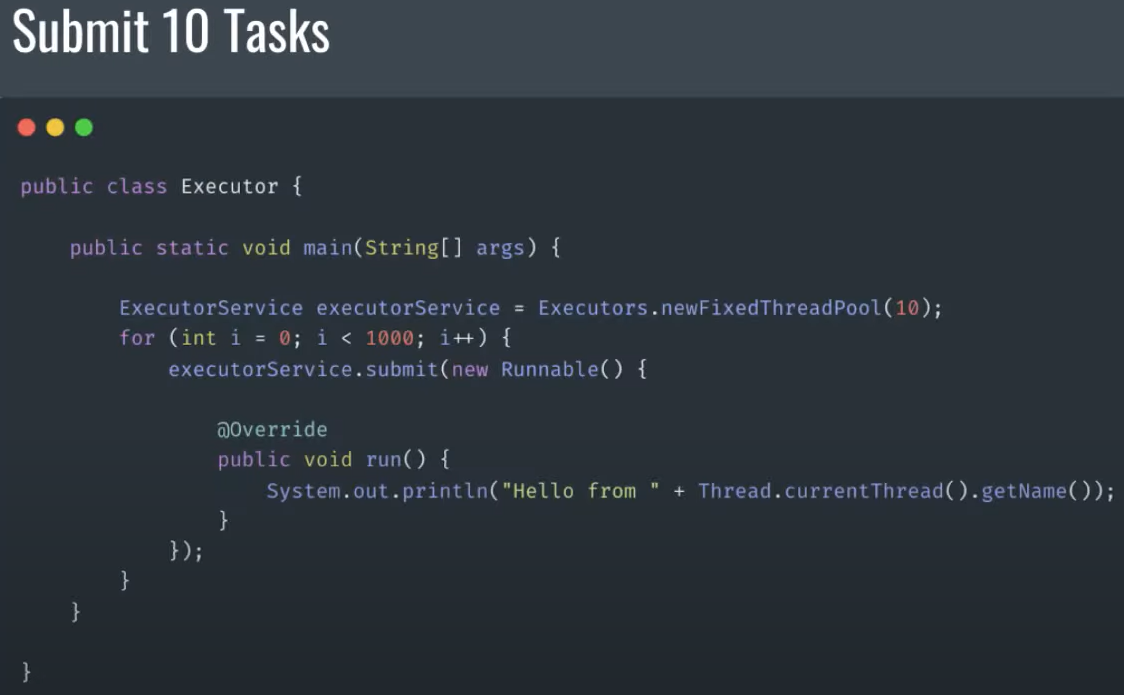
Bu tarz durumları cözmeye yarayan bilesenlere **Executor Service** diyoruz.

**Executor Service**

A Java interface that allows us to execute **tasks** on **threads** asynchronously.

Yukarıda for döngüsü icinde thread yaratırken bir runnable var yani lambda o aslında **tasks’ın kendisi. Bu tasklar threadde calısıyor.**

**Yani for icindeki Sysout kısmı bir task yani runnable. Üstündeki new Threadde threaddir, yani o taskı o thread’in icinde calıstırıyor. Thread bunun yüklenicisi. Bunlar asenkron sekilde oluyor.**



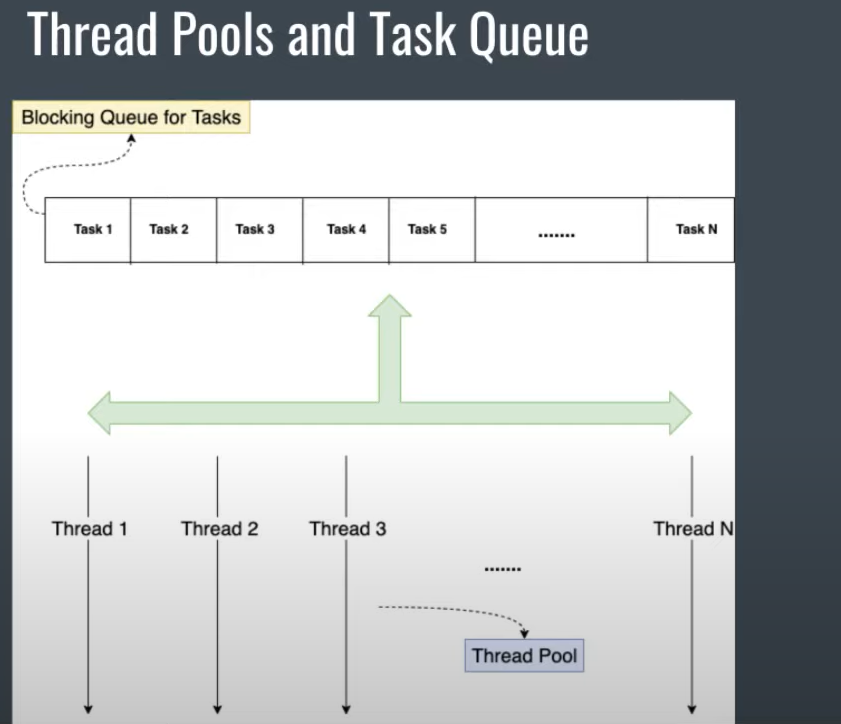
Burada ExecutorService tanımlayıp newFixedThreadPool(10) dediğimiz zaman burada elimizde 10 tane thread oldugunu belirtiyoruz.

Gerçek hayattaki örnekte ise 10 tane calısan oldugunu belirtmis olduk.

Executors icinde farklı threadPool mekanizmaları bunlardan bir tanesi de newFixedThreadPool’dir. Bu da fix yani sabit değer alıyor ve onu belirliyor anlamında.

Burada executorService icine task submit ediyoruz threadPool size belli,

O zaman bu durumda su olacak, gereksiz yere 990 tane thread yaratmayacak, burada threadin size’nin 10 olduğunu sabitliyoruz. Calısmamasına rağmen bircok thread olmayacak. Bu threadler taskları bitirmeye calısacak.



Thread Pool kısmı yukarıdaki örnekte newFixedThreadPool(10) kısmı burada n tane thread verebiliriz. Orayı temsil ediyor.

Yukarıda iste taskı submit ediyoruz ve taskı alıp Queue icine koyuyor arkada ve onun icin de alttaki threadler taskın icinden elemanları alıp alıp isliyor. Bosa cıktıkca yenisini alıyor bosa cıktıkca yenisini alıyor.

1.Taskı alıp bitirince diğerini alıyor. Success alıp bitirince alıyor. Kısaca (message queue mantığı diyebiliriz.)

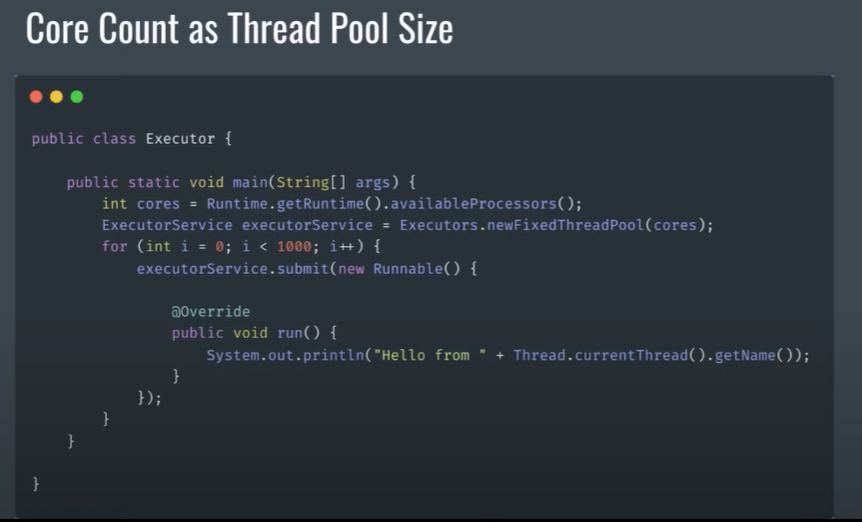
**Thread Pool Size**

If the tasks are **CPU intensive**, you can set pool size as **CPU Core Count**

If the tasks are **IO intensive**, then you can set your pool size as **some large value**

Eğer yaptığımız islem **CPU** **intensive** yani **CPU’yu** ac bi task ise o zaman **CPU Core count** kac ise thread pool sayısını core counta göre vermeliyiz.

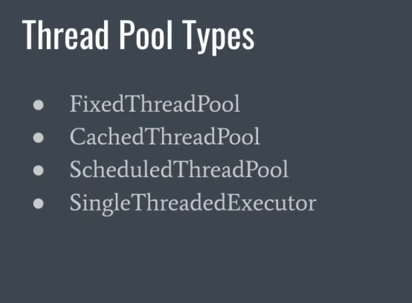
CPU intensive olduğunu da use case göre belirlenir. Degisebilir.



Core count’I anlama kısmı ise, int cores deki kod ile alabiliyoruz.

Buradaki tek fark ise daha önce 10 veriyorduk ama burada core sayısını buluyoruz ve kac tane core varsa o kadar count veriyoruz.

Thread Pool’ların Dört tipi vardır.



**FixedThreadPool**

We already visited this, it is a thread pool type that you define pool size, and it can the maximum size of threads that you can create. They will consume tasks from a **LinkedBlockingQueue**

Bir kere tanımlıyoruz. Sabit değer gibi düsünebiliriz. Burada taskı submit ediyoruz ve gittiği veri yapısı ise **LinkedBlockingQueue’dır.**

**CachedThreadPool**

There is no initial size of this thread pool, by default it starts from **0** and can go to **Integer.MAX\_VALUE.**

It uses **SynchronousQueue** which causes having only **one task** at a time

**Short** **running** **task** is the best one for this kind of pool.

Devamlı bir task submit ediyoruz, o zaman da thread pool 0’dan baslıyor ama sanki yapıyormus gibi yapar.

Yani burada yeni bir task submit ettiğimiz de yeni bir thread yaratmadan önce bu sistemde daha önce ki tamamlanmıs threadlere bakıyor, aynı islevi bir daha yapabilir mi diye kontrol ediyor. (Normal bildiğimiz cache mantığı ile aynı varsa cache’ten kullan yoksa dbden çek gibi)

Burada **CachedThreadPool ‘da yaratılan** threadlerin 60 saniye süresi var. Burada sistem su sekilde diyelim ki thread-2 10 saniyede isini tamamladı geri kalan 50 saniye de bos durmuyor, ben is yapabilirim mantığı ile duruyor ve task submit edildiğin de yeni bir thread yerine 50 saniyesi kalmıs thread pool icindeki threadi alıp onu isliyor.

Burada **SynchronousQueue kullanıyor. Çünkü** aynı anda bi tane task submit edebiliyor. Aynı anda sadece bir tane task var çoklu şekilde eklenmiyor.

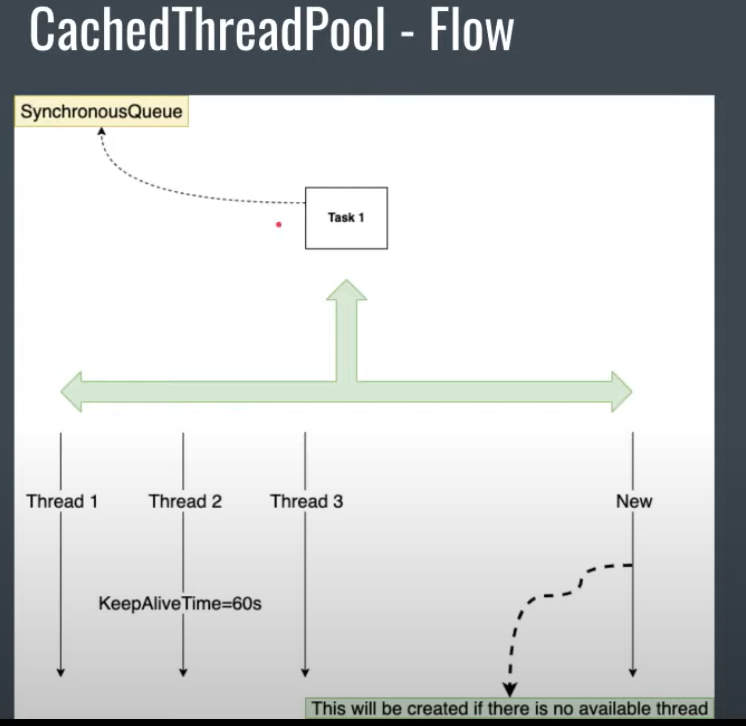
Burada **thread pool’ın boyutu da farklı 0’dan Integer.MAX\_VALUE’ye** kadar gidiyor.

Eğer kısa süreli calısan tasklarımız varsa bu thread pool cok iyi ve kullanıslı.

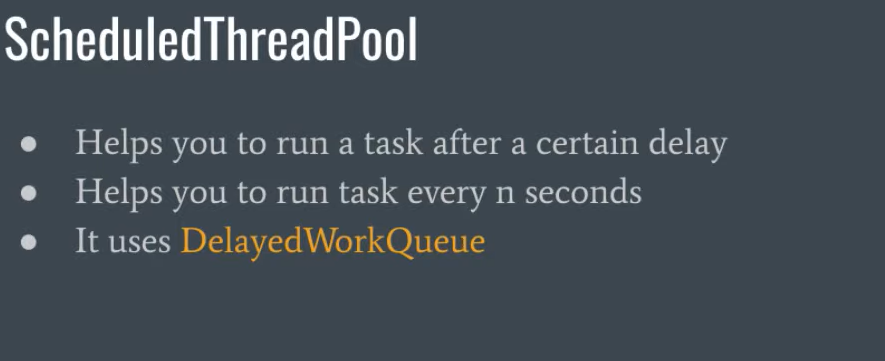
Cünkü bir task submit ettik, 10 sn icinde bitti o yaratılan threadi baska is icin de kullanabiliyor olmamız.

**Tasklarımız uzun sürüyorsa bunu kullanmamalıyız.**

Burada starvation olur, yani her zaman yeni bir thread yaratılacak var olanı kullanamayacak bir zaman sonra da islemci cevap veremeyecek. Buradaki threadlerin süresi zaten kısa.



Yukarıda gözüktüğü gibi tasklar kısa sürüyorsa 3 threadle taskları halledebiliyor,



Springteki @Scheduled calıstırılan taskların su kadar sürede, baslangıcta su kadar delay olsun mantığında düşünebiliriz.

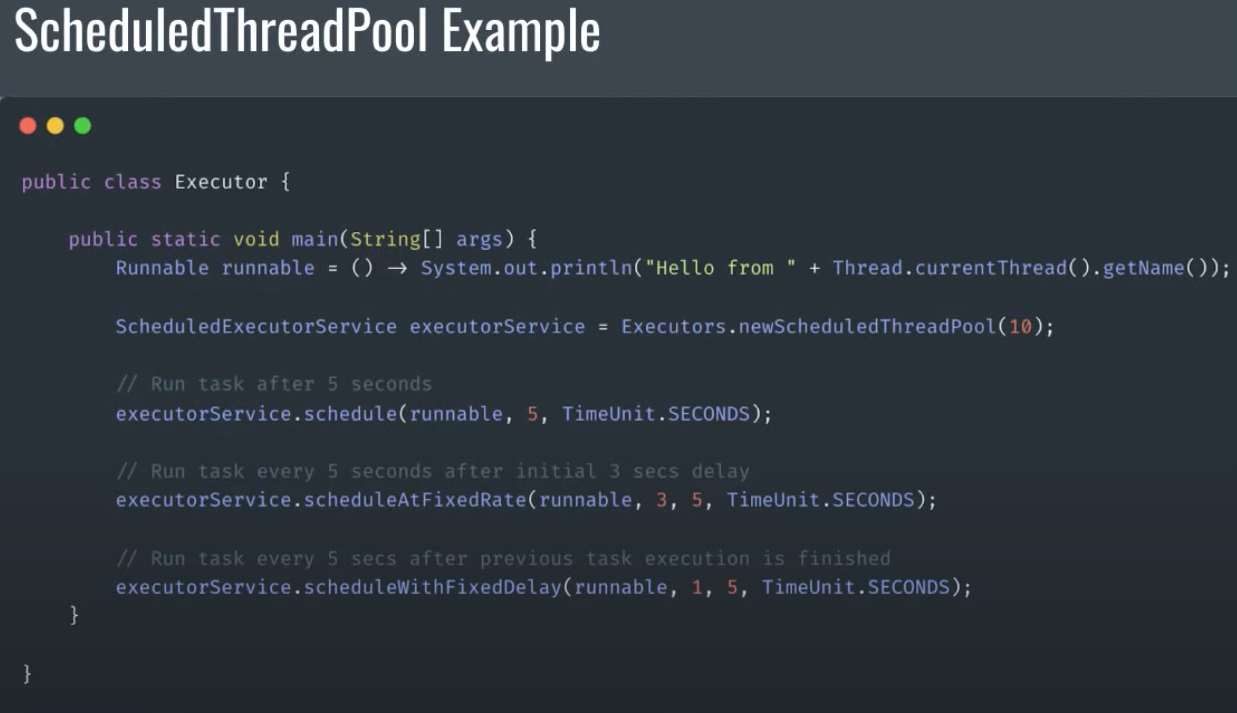
Burada iki durum var.

Bir taskı submit ediyoruz ama delaylı yani 15 sn sonra calıssın gibi bir mantıkla calıstırıyoruz.

Örnek olarak uygulamamız diyelim 15 sn kalkıyor biz gidip uygulama ayağa kalktıktan 30 sn sonra threadi calıstır diyebiliriz.

Sadece bir kereye mahsus değil de 5 sn’de bir calıstır diyebiliriz. Yani belli aralıklarla sürekli calıstırabiliriz.

Burada ise **DelayedWorkQueue** veri yapısı kullanılır.

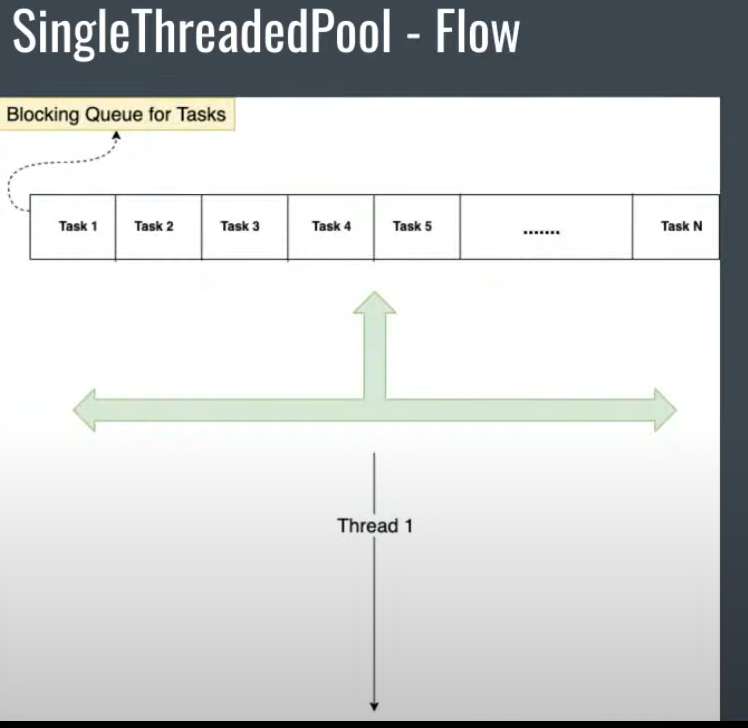


**SingleThreadedPool**

There is only thread consuming task queue

**LinkedBlockingQueue** is used.

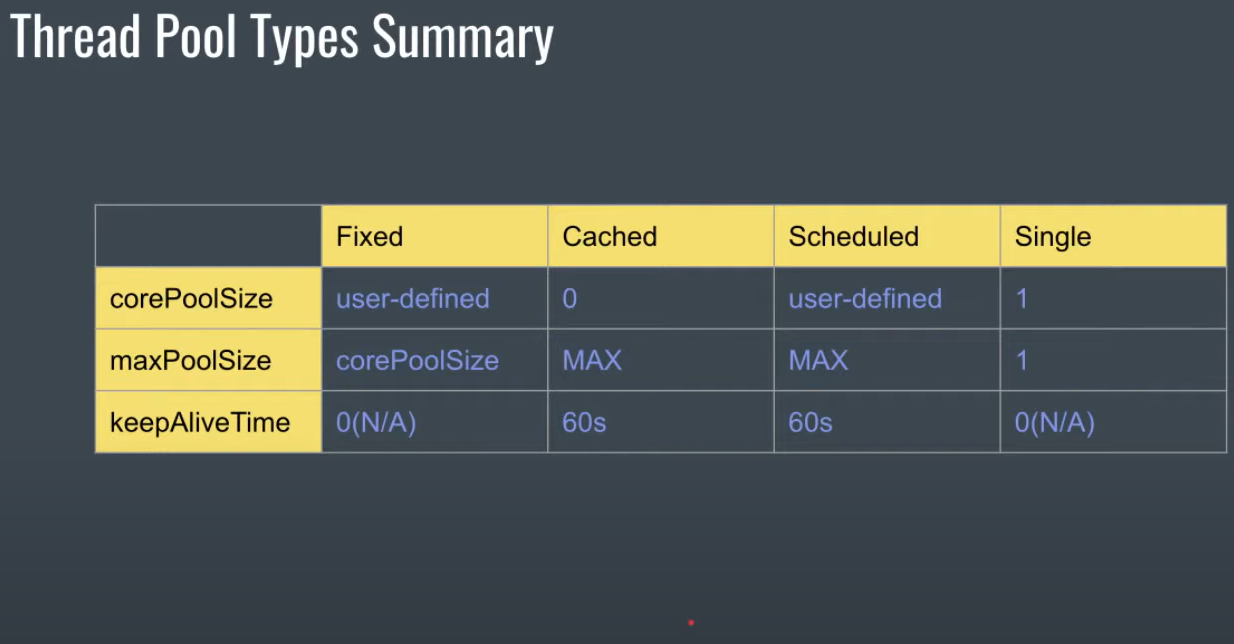
Threadpool icinde sadece 1 tane vardır. **LinkedBlockingQueue kullanır.**

****

Task sayısı birden fazla olabilir.

Paylaşımlı hosting gibi düşünebiliriz.

Burada usecase olarak müsteriye özel executor servisler açıp her birine bir thread açıp birden fazla task olabilir ve kendilerine ayrı life cycle ları olabilir.



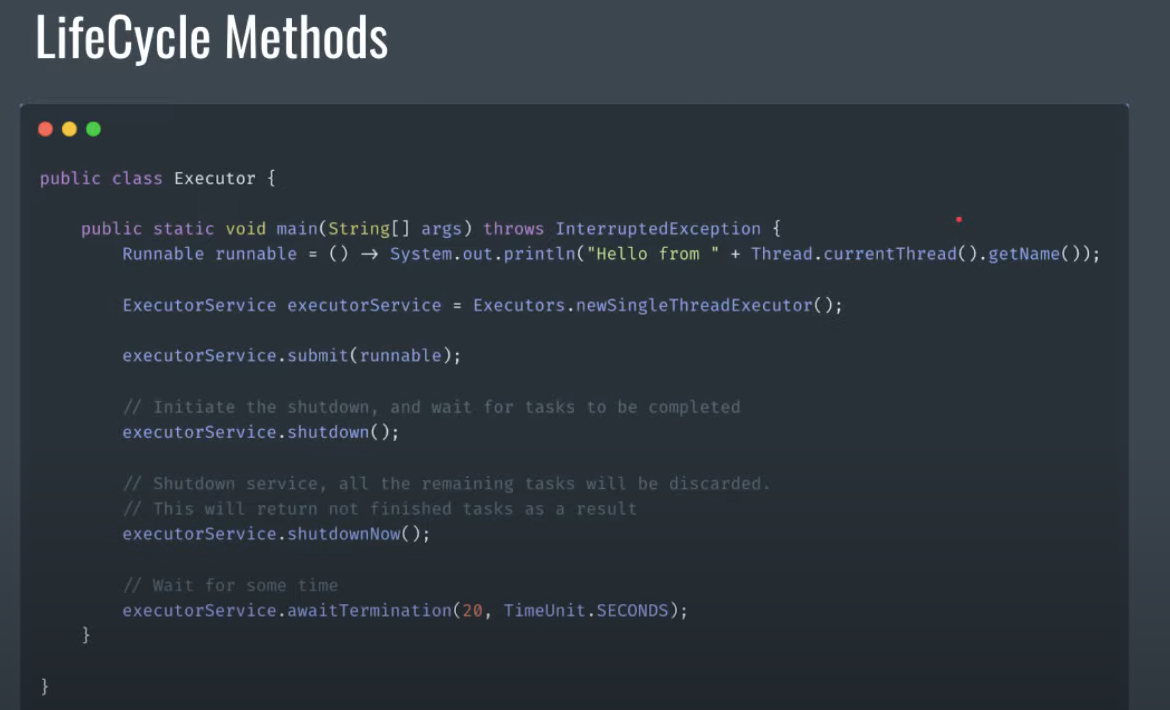
Bu dört tipte hepsine gecilen 3 parametre var.

User-defined bizim verdiğimiz değerlere karsılık geliyor.

corePoolSize ne ise maxPoolsize de odur.

keepAliveTime threadin yasam süresi. Yasam süreleri degistirilebilir.

Executor servisi yaratttığımız da onun üzerine bir seyler yapabilir miyiz sorusuna cevap lifeCycle.



**Shutdown** dediğimiz zaman islem kapanmaya baslıyor ama icinde calısan islemler bitince öyle kapanıyor.

**shutdownNow** dersek iceride ne varsa islemlerin hepsini öldürüyor.

**awaitTermination** 20 sn bekliyor kapanmasını 20 sn bekledikten sonra timeout diyip yoluna devam ediyor.

**Callable vs Runnable**

Runnable helps you to run task, does not care about return value

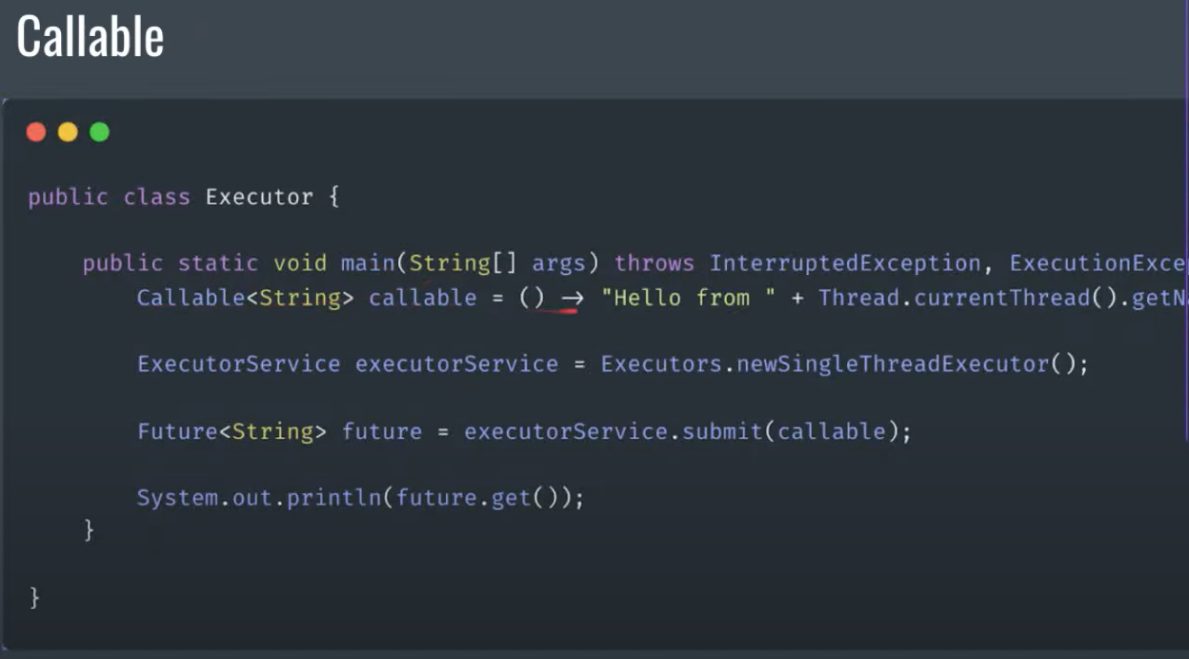
Callable helps you to get result for submitted task. It returns a **Future** that you can get in blocking way.

**Runnable** void oldugu icin cevabını alamıyoruz.

**Callable** de ise submit ettikten sonra bize feature dönüyor, o feature’n ne zaman biteceğini bilmiyoruz. Almak istiyorsak da feature.get diyip onun üzerinden alamıyoruz.

**Runnable v**

**e Callable farkı da o birinde cevap alamıyoruz, birinde alıyoruz.**

****

51.00 de kalındı.