**Process**

Collection of code, data, memory, and resources. This is typically an application.

Process aslında bir uygulama.

Process code, data, memory belli zamanlarda diske erisiyor, resources bunların hepsinin toplamı bir uygulama bu aslında bir process

Process : isletim sistemi icinde kendi basına calısan tamamen izole olan uygulamaladır. asagıda acıklaması var.

mesela autocad uygulaması actık, bi de chrome uygulaması actık chrome memory’de yer tutuyor, yani autocad uygulamasının chroma ait memorye erisim yapıp orada ona ait bir sey alıp kullanması mümkün değil yani tamamen birbirlerinden izole.

Yani process’in temel mantığı bu kendine bir alan yaratıyor ve onun icinde calısmaya baslıyor.

**Thread**

Sequence of code which is executed in scope of process.

**Thread** dediğimiz sey process’in altında olan bir sey.

**Thread** dediğimiz sey de contexti process’e cekiyoruz.

Process icinde code’lar var demistik burada codeların icindeki mantığı asıl calıstıran mantığa **thread** diyoruz.

**Thread** aslında yük tasır yani verilen emri yapar.

**Process bir islem izole sekilde calısan, onun altında threadler vardır asıl is yükünü götüren.**

**Concurrency**

The task of running and managing multiple computations at the same time.

PC’de CPU vardır, CPU asıl computations yapan islemdir.

**Concurrency :** Birden fazla isle muhattap olma islemine deriz.

O ortamda büyük bir isin kücük parcasını yapıyoruz, yani belli bir is yaparken beklemek zorunda değiliz o arada baska bir islem de yapabiliriz, ek is gibi düsünebiliriz.

Calısan bir mekanizma olarak bir thread %100 islem yapmıyor yani söyle bir thread bir islem yaparken baska bir thread bos buldugu bir anda baska bir islem yapabilir.

**Parallelism**

The task of running multiple computations simultaneously.(eş zamanlı)

Burada aynı anda aynı zamanda birden fazla isi paralel olarak yapıyor. Eş zamanlı bir şekilde.

Paralellikten bahsetmek istiyorsak CPU üzerinde birden fazla çekirdeğin olması lazım.

Aynı anda iki tane farklı threade CPU icinde cekirdek 1 ve cekirdek 2 ‘ de calıs, aynı anda aynı zamanda islemleri yapabiliriz.

**Concurrency vs Parallelism**

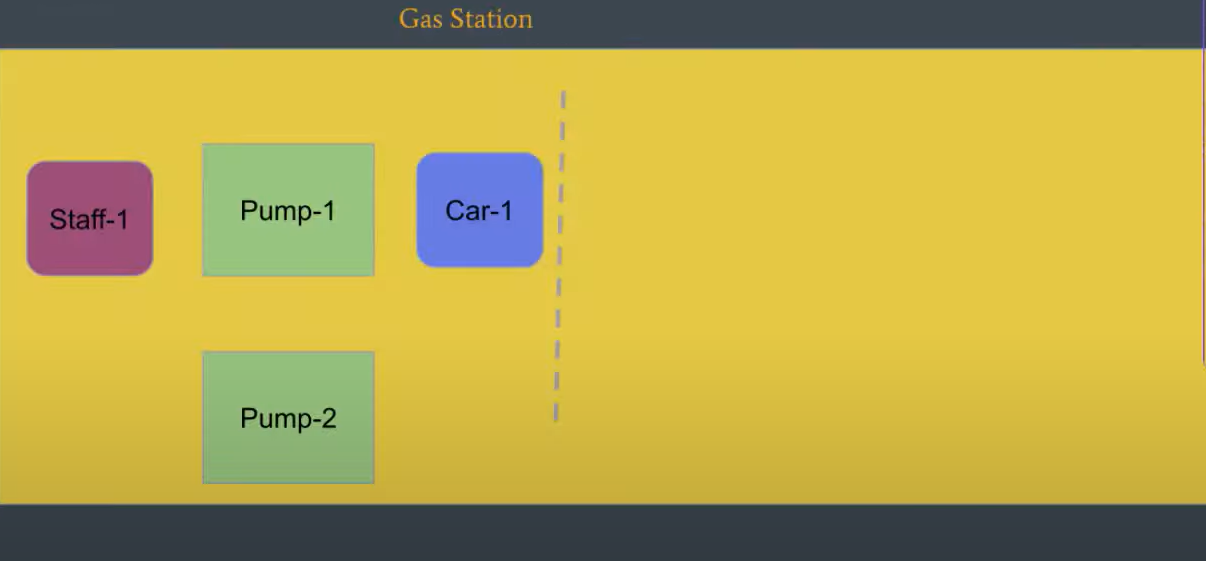
Concurrency is **dealing** with lots of thing at once while Parallelism is **doing** lots of things at once

Concurrencyde dealing yani birden fazla is ile muhattap olma, ama gercekten aynı anda iki is yapmıyoruz. Burada thread 1 is yapıyor arada bosluk olma durumunda thread1 baska is ile ugrasırken thread 2 calısıp isini hallediyor.

**RealEx** : Gerçek hayattan örnek verecek olursak, benzin alırken benzin dolarken (thread1) o sırada gidip kasaya ödemeye yapıyoruz(thread2). Burada thread1 calısıyor, thread1 benzinin dolumu ile ilgileniyorken thread 2 o boslukta ödeme islemini tamamlıyor. Dolayısıyla birden fazla is yapıyoruz ama aynı zamanda değil yani hem benzin pompasını takarken hem de kasaya gidemeyiz.

Parallelism de ise iki thread iki de core(cekirdek) olması gerekiyor cünkü aynı anda aynı zamanda gerceklesiyor.

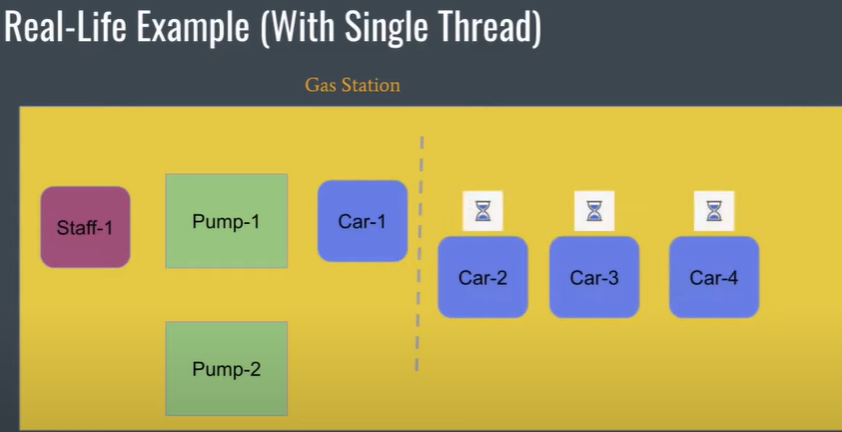
**RealEx** :Gerçek hayattan örnek olarak ise benzin istasyonundaki doldurma yerlerinden birine core1 diğerine de core2 dersek iki islem yani iki araba geldiğinde biri core1 biri de core2 de aynı anda eş zamanlı benzin dolum islemi gerceklesebiliyor.

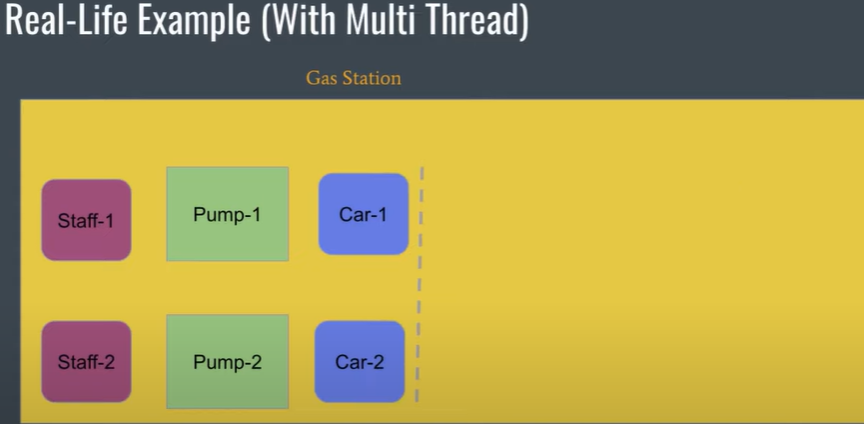
**Real-Life Example ( With Single Thread)**

**Staffları Thread, pumpları core(cekirdek) olarak düsünelim.**

Case’de 2 tane benzin dolum yerimiz (pump), bir tane calısan (Staff)

Simdi 1.araba geldi bir tane de calısan var, ve o car-1’e hizmet verecek.

Dolayısıyla asagıdaki gibi diğer arabalar geldiğinde beklemede olacaklar. Cünkü elimizde bu isi yapabilecek bir tane eleman var o da Staff-1. **Yani bir calısan var bir thread var. Yani paralelde gidip baska bir is yapamaz.**

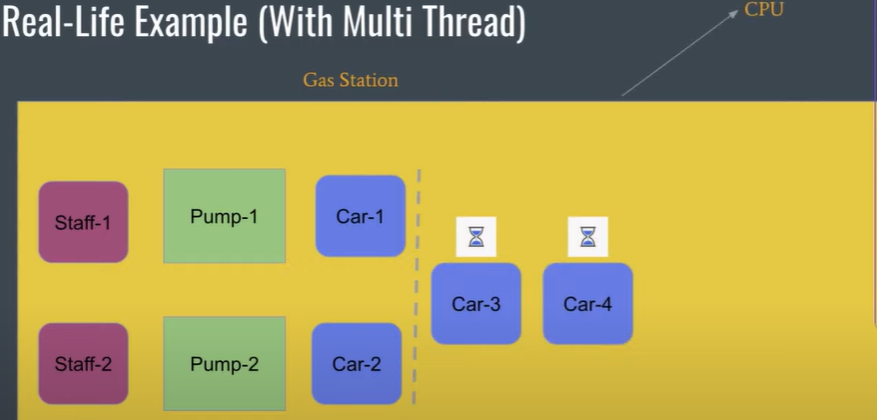
**Real-Life Example ( With Multi Thread)**

**Staffları Thread, pumpları core(cekirdek) olarak düsünelim.**

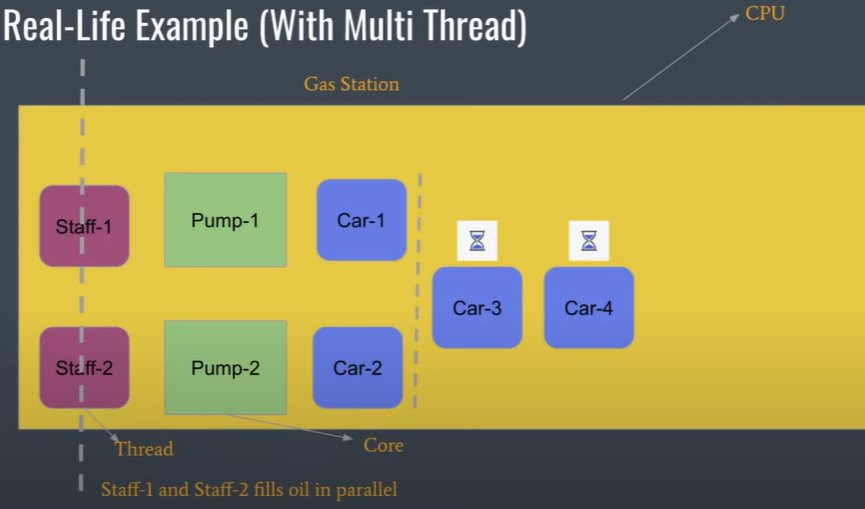
2 tane core vardı zaten, 2 tane de thread(staff) var.

2 araba bir anda geldi 2 thread var zaten 2’si de islemini halledebilir. **Zaten bu multithread’dir.**

**Ama** asagıda gözüktüğü gibi 3 ve 4. Arabalar geldiğinde bunlar bekleyecek.

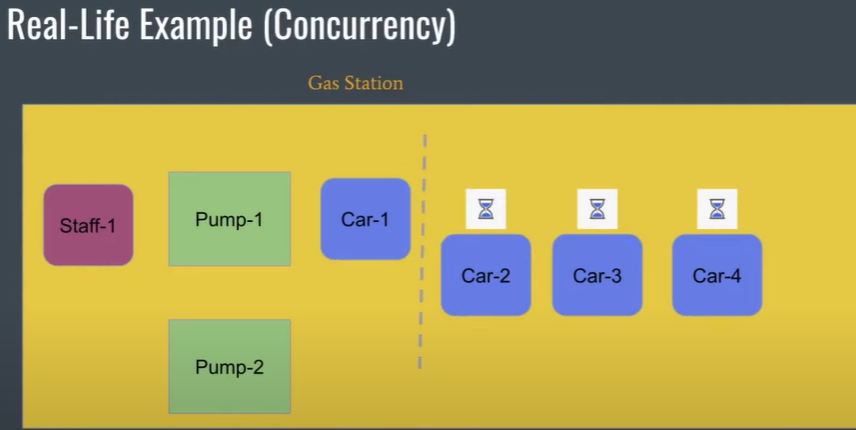
****

**Nihayetinde baktığımız zaman benzin istasyonu bir CPU’dır.** Çünkü yukarıdan baktığımız zaman icinde birden fazla islem var,(kasa kısmı, su yıkama, hava dolduran kısımlar vs.)

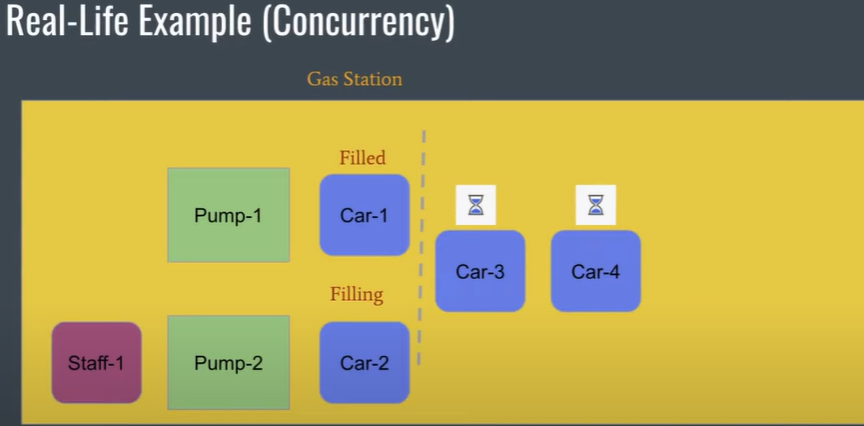


Yukarıda baktığımız zaman staff-1(thread1) ve staff-2(thread2) aynı anda eş zamanda car-1 ve car-2 ye hizmet vererek parallel islem yapmıs oluyor.

**Real-Life Example (CONCURRENCY)**



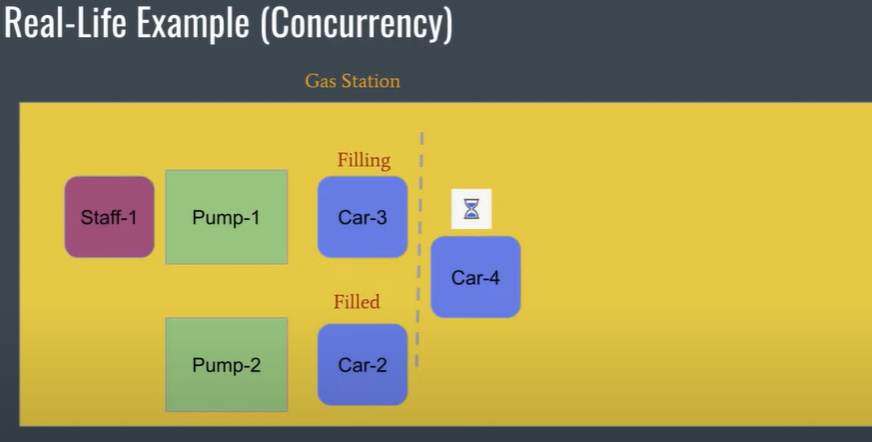
Thread1 yani staff-1 pompayı taktı car-1 dolum yapıyor. Diğerleri bekliyor.



Car-1 doluyorken thread1 gitti car-2 yi aldı ve core2 de onu doldurmaya basladı. Neden thread bir tane var dersek burada neden car-2’yi aldı baska bir islem aldı dersek yukarıda bahsettiğimiz bosluk tam olarak bu. Car-1 dolum yaparken thread1 oradan ayrılabilir o boslukta baska bir islem alıp yani car-2 ona islem yapıyor.

Aynı islemler asagıdaki gibi devam ediyor bu sefer car-1’in islemi bitiyor car-2 dolarken car-3 ü alıyor car-2 bittiğin de ise car-4 alacak bu sekilde ilerleyerek thread islemlerini bitirecek. **Yani burada temel anlamamız gereken sey bir tane görevli birden fazla sey ile muhattap olabiliyor. Buna concurrency diyorıuz ama bunları parallelde yapmıyor.**

**Parallelde yapabilmesi icin iki thread yani iki staff oldugu zaman yapabiliyor. Ona da parallelism diyoruz.**



**Java Threads**

JVM is a process and it spawns threads based on the implementation. It starts with main thread, and it can spawn another threads based on customer request

Code yazıyoruz bu kodun icinde farklı thread yarat demediğimiz icin bizim icin thread yaratılmaz. JVM bir process’dir.

Bu process farklı threadlari acıp yönetebilir kodumuza göre.

Public static void main dediğimiz zaman uygulamanın giris noktasıdır ve uygulama burada calısmaya baslar. **Main thread**

**Main thread ile baslıyoruz kodumuza göre iceride yazdığımız koda göre birden fazla farklı thread acabiliyoruz.**

En güclü örneği olarak servlet dediğimiz mantıklardır.

Spring’de uygulamayı ayağa kaldırdığımız da bir request geldiği zaman arkada db’ye bağlanıyor farklı islemler yapıyor, o sırada request2’ye alamam gibi bir durum söz konusu olmuyor request3’ü de alıyor dördü de.

Bunu bize sağlama kısmı ise icerideki servlet logici requesti aldıktan sonra hemen bunun icin ayrı bir thread acıp bu threadde islemlere devam ediyor. Bu sayede birden fazla requesti bu uygulamara atabiliyoruz.

Ana request baslar onun icinde servlet logici vardır. O servlettin yönettiği ve asagıda acılan bir thread dallanması vardır. Her requeste karsılık gelen yeni bir thread acılır.

Request gelip islemi bitirip kendini kapatıyor.

**Stack vs Heap**

Method call chain performed within thread is stored in stack with method arguments and local variables.

All the objects that are created in the application are stored in the heap which is shared space

Threadler icin is yükünü sırtlanan demistik. Threadler kendi icinde bircok is yapıyor method cagırıyor, fonksiyon cagırıyor vs dallanıyor sonuc geldikce kendini geriye toparlaması lazım. Dolayısıyla bu cagırdıgı seylerin history’sini tutabilmek icin method arguments and local variables vs. stack’ kısmında tutuyor.

**Thread’lerin kendi özel kapalı bir kutuları var ve onun icinde stackte tutuyor. Kendi icindeki stackte tutuyor yani.**

Thread’ler bir seyler yarattığı zaman ya da bir thread’in icinde baska bir servise bağımlılığı var. Yani category servicede thread actık diyelim o da product service kullanıyor yani bir referans var bu tarz referanslar **yani yaratılan objelerin kendileri heap’te tutuluyor**.

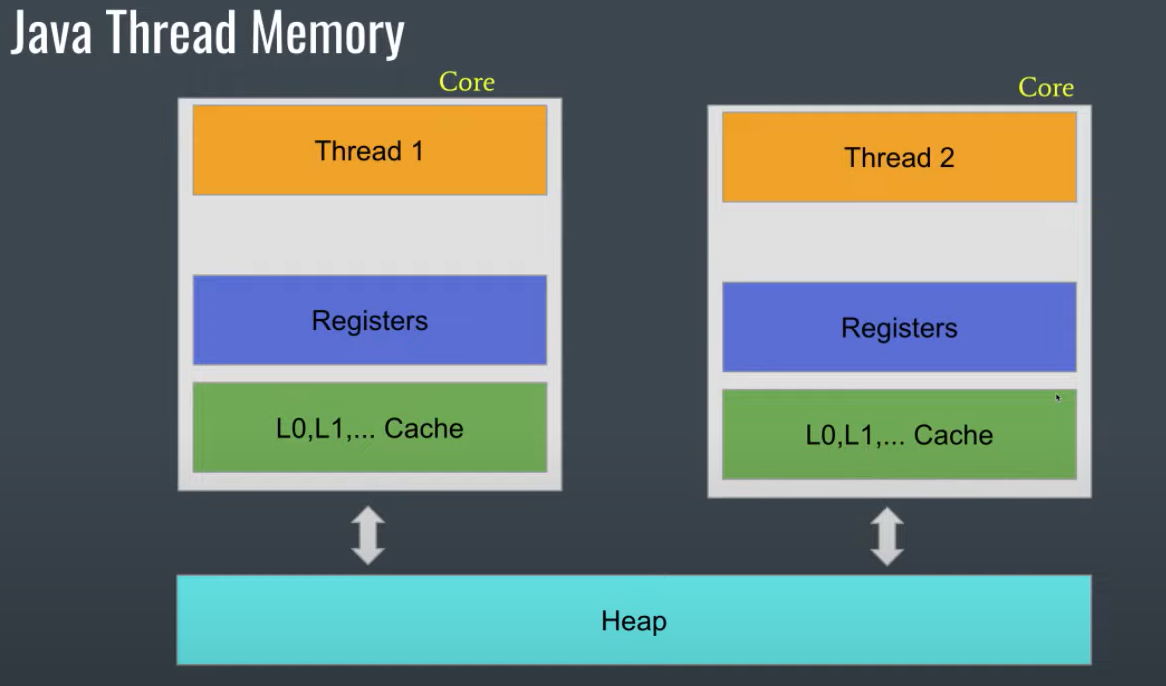
Bunu bir RAM’de duruyor diyebiliriz. Ram dediğimiz sey JVM process icinde heap dediğimiz alan uygulamanın tamamına görünür.

Yani JVM icinde obje yaratılırken o heapte tutuluyor o objeye baska yerden ulasabiliriz.Farklı bir threadden ulasabiliriz. Ama referanslar thread’lerin icinde duruyor.

Yani örnek verecek olursak bir müsterinin birden fazla bilet alma islemi olabiliyor ve bir hesap düsünelim aldıkca o hesaptan düsecek, burada diğer threadler ulasarak hesaptan islemi düsebiliyor.

Objenin kendisi heapte ama nereden ulasması gerektiği yani referansı ise heap icindeki stack’te tutulur.

Stackte genelde pratik bilgiler tutulur. Kısa dönemde yasayan, referans olan. Asıl verinin kendisi Heapte durur.



Thread calısırken kendi icinde cache mekanizması var. Veriyi tutuyor.

Daha sonra heap’e ne zaman yazılacağı ilk basta bizim elimizde değildi. Burada söyle bir sıkıntı var diyelim bakiye 100 tl thread1 70 tl islem yaptı, thread2 de 50 tl, thread1 islemi bitirdi bakiyeyi 30’a düsürdü thread 2 de düsürecek ama burada sorun var thread2de bakiye 100 gözüküyor cünkü ortak bir datayı kullanıyorlar. **Bu bir visibility(görünürlük)sorunudur.**

Burada eger thread1 herhangi bir veriyi güncellediği zaman bunu heape yazsaydı ve thread2 tarafından da erisilebilir olsaydı ikisi de ortak data üzerinden islem yapacaktı, yani yukarıdaki gibi bir islem değil de thread1 bakiye güncelleyip heape yazdıgın da thread2 de eski veriyi değil de güncellenmis veriyi görebilirdi. **O zaman da visibility ( görünürlük) sorunu kalmayacaktır.**

Bunu yapmak icin ise Java’da “**Volatile**” keyword vardır.

**“Volatile” Keyword**

Thread has its own local cache, so when you update something, it will be flushed to heap after some unknown time period.

To replicated it directly to heap, we use **volatile** keyword.

Thread1 islem bittiğinde bakiyeyi düsürdüğünde bunu anında heape yansıtıyor. Dolayısıyla da thread2 de bu veri güncellenmis oluyor.

**Heap’e direkt olarak yansımasını istiyorsak volatile kullanmamız lazım.**

**Volatile sadece visibility sorununu cözüyor, datanın kirli olmasını cözmüyor.**

**Synchronization in ise hem visibility, hem de datanın kirliliğini önlüyor.**