Abstract Classes

* Abstract methodlar (c++ de pure virtual function) miras alınmış sınıflar tarafından implemente edilmesi zorunlu olan fonksiyonlardır.
* Bir fonksiyonun override edilmesini ***final*** anahtar kelimesiyle kapatabiliriz. “public final String … “

Interfaces

* Bir interfacede sadece method imzaları bulunur ve belirtilen attributeler ***public static final*** şeklinde otomatik olarak oluşturulurlar. Methodların bodyleri bulunmaz.
* ***Default method*** ile bir interface içerisinde body ile beraber method oluşturulabilir ayrıca bu method implement edilen classda implemente edilmek zorunda değildir
* ***Default methodlara*** yardımcı method oluşturmak için ***concrete*** ***private***, **concrete *private static*** veya farklı bir amaç için **concrete *static*** helper method oluşturulabilir.

Interfaces Vs Abstract

* İkisinden de instantiate yapılamaz
* İkiside concrete methodlar içerebilir (interfaceler JDK 8 den sonra bu özelliği kazandı)

metin, ekran görüntüsü, makbuz, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

Generics

* Generic class tanımlarken “**public class** className<T **extends** otherClassName>” extends ile generic classın kabul edeceği classların sınırlandırması yapılabilir.
* Benzer bir sınırlandırma da şu şekilde yapılabilir;metin, ekran görüntüsü, yazı tipi, yazılım içeren bir resim

  Açıklama otomatik olarak oluşturuldu
* Bir fonksiyona parametre olarak generic tipte bir değişken yollanacaksa bu değişkenin upper ve lower boundu belirlenebilir. metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

  Açıklama otomatik olarak oluşturuldu



* List<?> ile yalnızca List’in farkı; List kullandığımızda geçirilen değişkene fonksiyon içerisinde her türde eleman eklenmesi yapılabilir Integer, String vs, <?> kullandığımızda bunu kapatıyoruz. Liste içerisinde değişiklik yapılamıyor.
* Methodlara parametre olarak generic tip gönderildiğinde, runtime da bytecode’a veya class dosyasına dönüştürülürken tip parametresi es geçilip upper veya lower bound varsa onunla yoksa object ile değiştirilir.

Örneğin   




Şeklinde overload yapıyor olun, bu hata olarak alınır çünkü iki fonksiyonunda signature’u aynıdır. Yani dönüşüm yapılıyorken bu senaryoda bound olmadığı için ikisi de



Şeklinde ele alınır. Bu işleme ***type erasure*** denir.

* Aşağıdaki gibi upper bound olarak birden fazla tip belirtilebilir. Bu tiplerin hepsinin T tarafından implemente edilmiş olması gerekmektedir

metin, ekran görüntüsü, tasarım içeren bir resim

Açıklama otomatik olarak oluşturuldu

Inner Class

* Bir inner classdan nesne üretilmek isteniyorsa önce enclosing class’dan bir instance’e sahip olmak gerekir.

metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

Açıklama otomatik olarak oluşturuldu

**veya**



Local Class

* Bir method, döngü veya if deyimi gibi bir yürütülebilir blok içinde tanımlanırlar. Enclosed kısma erişebilirler. Yerel sınıflar, dış değişkenlerin değerlerini kopyalar ve bu kopyaları kullanır. Bu nedenle, dış değişkenlerin değeri değiştiğinde local classdaki kopyada değişiklik olmaz bu da beklenmeyen davranışlara yol açabilir. Bu yüzden local classda final veya effectively final dış değişkenlere erişime izin verilir.

metin, ekran görüntüsü, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

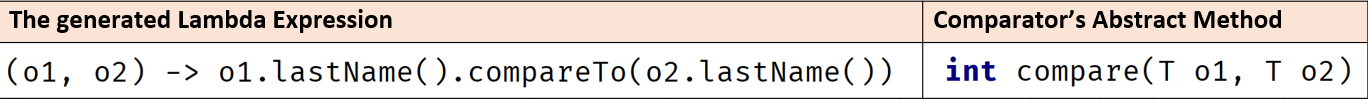
Anonymous Class

* Java’da, bir anonim sınıf, ismi olmayan bir iç sınıftır. Anonim sınıflar, kodunuzu daha kısa hale getirmenizi sağlar. Aynı anda bir sınıfı bildirip örnekleyebilmenizi sağlar. Yerel sınıflara benzerler, ancak isimleri yoktur.
* MyRunnable interface’sini/sınıf’ını kalıtım alan ve onun run() methodunu override eden bir Anonymous class örneği

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

Lambda expression



* Lambda expressionların kullanımı C++’deki gibi değildir.
* Bir lambda expression kullanmak için, bir functional interface’e ihtiyaç vardır.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, yazı tipi, ekran görüntüsü, tipografi içeren bir resim

Açıklama otomatik olarak oluşturuldu

* Funcitonal interface’in abstract methodu lambda expression olarak implemente edilir.
* Ek olarak farklı yerlerde kullanabilmek için anonymous class olarak da functional interface implemente edilebilir.

metin, ekran görüntüsü, yazı tipi, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

* Anonymous classlarda olduğu gibi lambda expressions’larda da enclosing scope’daki değişkenlere erişilebilir fakat ***final*** veya ***effectively final*** olması gerekir.

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

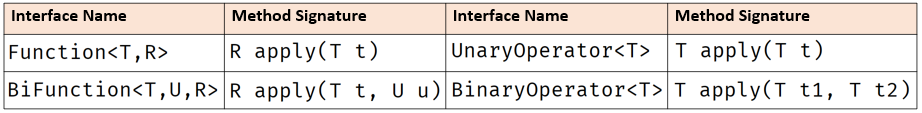
* Javada functional interface’lerin bulunduğu java.util.function package’i bulunmaktadır.

İçerisindeki bazı functional interface’ler ve kullanımları;

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

* Function interface’in farklı türleri, return edilecek değer hep en sonda yazılır, yani aşağıdaki ***R*** olanlar



Method references

• Lambda ifadesinin daha kompakt ve kolay bir formudur. Her seferinde lambda ifadesini sadece bir metodu referans almak için kullandığınızda, lambda ifadenizi method reference ile değiştirebilirsiniz.

**metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

Açıklama otomatik olarak oluşturulduMethod references tipleri**

metin, ekran görüntüsü, sayı, numara, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu**metin, ekran görüntüsü, sayı, numara, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturulduÖrnekler**

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

Comparator’s additional helper methods

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

Collections

metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

Açıklama otomatik olarak oluşturuldu

Set

* Tek bir null element içerebilir.
* Duplicates bulunmaz
* Sıralı değildir

HashSet

* + Duplicates kontrolu için comparable implemente edilmesi gerekir.
  + Elemanlar dağıtıktır

TreeSet

* + Red-black treedir
  + Sıralıdır

metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

Map

* map işte

view

* verinin tamamını kopyalamadan veya yeni bir veri yapısı oluşturmadan, mevcut verinin belirli bir görünümünü sunar. Bu, veriye erişimi kolaylaştırabilir ve performansı artırabilir. Veriye yapılan değişiklikler, genellikle orijinal veri yapısını etkiler, çünkü aslında aynı veriye işaret ediliyor. Ancak, bazı view türleri sadece okuma erişimi sağlayabilir ve değişiklikleri yansıtmayabilir.

Örnekler:

**SubList View (Alt Liste Görünümü):** List koleksiyonlarında subList() metodu kullanılarak elde edilen alt bir liste, orijinal listenin bir görünümünü temsil eder.

**MapView (Harita Görünümü):** Map koleksiyonlarındaki keySet(), values(), veya entrySet() metotlarıyla elde edilen görünümler, haritanın sırasıyla anahtarlarını, değerlerini veya anahtar-değer çiftlerini temsil eder

**Read-Only View (Salt Okunur Görünüm):** Bazı koleksiyon türleri, sadece okuma amaçlı görünümler sağlar. Bu, orijinal veriyi değiştirme riskini azaltır.

EnumSet ve EnumMap

* Yalnızca enum türleri ile kullanılabilirler ve enum sabitlerini içeren bir küme verisini yüksek performanslı ve optimize edilmiş bir şekilde saklamak amacıyla tasarlanmıştır.
* 0 ve 1’lerden oluşan bit vectorleri ile ifade edilirler, işlemler bit math kullanılarak yapıldığı için çok hızlıdır aynı zaman az bellek tüketir. Verimlidir.
* RegularEnumSet 64 enum değerine kadar tutabilir, 64’den fazla eleman için JumboEnumSet kullanılır.

Immutable

* Java'da, immutable (değişmez) nesne, bir kez oluşturulduktan sonra içerdiği verilerin durumunun değiştirilemez olduğu bir nesne türüdür. Java dilinde bir nesnenin değişmez olması için aşağıdaki özellikleri taşıması gerekmektedir:

**Durumun Değişmezliği:** Bir nesnenin içerdiği verilerin (alanların) değerleri bir kez atanıp daha sonra değiştirilemez. Yani nesnenin durumu oluşturulduktan sonra sabit kalır.

**Alanların Özel Olarak Ayarlanması:** Alanların değerlerine doğrudan erişim olmamalıdır. Bunun yerine, nesne oluşturulurken veya özel bir yöntem aracılığıyla alanlar atanmalıdır.

**Değiştirilemez Yöntemler:** Değişmez nesnelerin iç durumunu değiştirecek yöntemler (methods) bulunmamalıdır. Yani nesnenin içeriği değiştiren herhangi bir yöntem olmamalıdır.

* Değişmez nesneler Java programlamasında birkaç nedenle tercih edilir:

**Thread Safety:** Değişmez nesneler birden fazla iş parçacığından güvenli bir şekilde erişilebilir, çünkü iç durumları değişmezdir. Bu yüzden senkronizasyon problemlerini önlemek için ekstra çaba sarf etmek gerekmez.

**Cache ve Performans Optimizasyonu:** Değişmez nesneler, aynı değerlere sahip nesneleri yeniden oluşturmak yerine paylaşarak bellek kullanımını optimize edebilir.

**Güvenilirlik:** Değişmez nesneler, yanlışlıkla veya kötü niyetli olarak içeriğinin değiştirilmesini engeller.

metin, ekran görüntüsü, yazılım, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

İlk bakışta xArgument’in, doXYZ methodu çağrıldıktan sonra değişmesi gerektiği düşünülüyor.  
Fakat String’ler immutable olduğu için değişmiyor.

Final Modifier

* Final instance method overridden edilemez
* Final static method subclasslar tarafından görülemez
* Final attributelere tekrar atama yapılamaz, constructorda, tanımlandıkları alanda, instance initializer blockta, global scope’da initialize edilebilirler.
* Final sınıflardan kalıtım alınamaz (kendi başka sınıftan kalıtım alabilir.)
* Final method parameter method içerisinde değiştirilemez.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

Bu koda parametre olarak “20” gönderildiğinde, çıktı 2015 olur. ***Final anahtar kelimesi yalnızca referansın değişmezliğini belirtir***. İçerik değiştirilebilir. X adında farklı bir değişken bulunsaydı z = X yapılamazdı.

Immutable Class

* İnstance alanları private ve final yap
* Setter method tanımlama
* Getterlarda defensive copies yarat
* Class’ı final yap veya tüm constructor’ları private yap

Method Hiding

* Java'da "method hiding" veya "method gizleme," alt sınıfların üst sınıflarda tanımlanan static metotları aynı imza (isim ve parametre listesi) ile tekrar tanımlamasını ifade eder. Bu durumda, alt sınıftaki yeni metot, üst sınıftaki metodu gizleyecektir. Ancak, instance (nesne) metotları için bu kural geçerli değildir, çünkü bu tür metotlar overriding'e tabidir. Static metotlar için overriding (geçersiz kılma) kavramı Java'da söz konusu değildir, çünkü static metotlar ve instance (nesne) metotları farklı davranışlara sahiptir ve static metotlar da dahil olmak üzere buna bağlı bazı nedenler vardır:

**Bağlama (Binding) Mekanizması:** Instance metotlar için dinamik bağlama (dynamic binding) gerçekleşir. Bu, hangi metodu çağırmanız gerektiğini çalışma zamanında belirler ve nesnenin türüne göre yönlendirme yapar. Buna karşılık, static metotlar için erken bağlama (early binding) kullanılır, yani hangi metotun çağrılacağı derleme zamanında belirlenir.

**Polimorfizm Yokluğu:** Overriding, polimorfizm (çok biçimlilik) konseptiyle yakından ilişkilidir ve instance metotların çalışma zamanında değiştirilmesine izin verir. Static metotlar için bu tür bir dinamizm yoktur ve bu nedenle polymorphic behavior (çok biçimli davranış) sağlayamazlar.

**Kapsam (Scope) ve Kalıtım Problemleri:** Eğer static metotlar overriding'e izin verseydi, alt sınıftaki static metot üst sınıftakini geçersiz kılabilir ve bu da kalıtım zincirinde kafa karışıklığına yol açabilirdi. Bu tür bir durum, hatalara ve mantık hatalarına neden olabilirdi.

**Static Metotların Sınıf Üzerinde Çağrılması:** Static metotlar genellikle sınıf düzeyinde çağrılır (örneğin, SınıfAdi.metotAdi() şeklinde). Alt sınıftaki bir static metot, üst sınıftakini geçersiz kılabilseydi, bu çağrılar beklendiği gibi çalışmayabilirdi.

Bu nedenlerle, static metotlar Java'da overriding'e tabi değildir ve bunun yerine method hiding (gizleme) kavramı geçerlidir. Yani, alt sınıftaki bir static metot, üst sınıftaki aynı adlı static metodu gizler (hider), ancak instance metotlar gibi dinamik bir bağlama ve geçersiz kılma söz konusu değildir.

* “BaseClass childReferredToAsBase = new ChildClass();” bu şekilde bir tanımlama yapılmış olsun ve çağrılacak aynı imzaya sahip bir normal bir de static iki method bulunsun. Overriding işlemleri dynamic binding olduğu için kod çalışırken childReferredToAsBase ChildClass’ın bir objesi olacağından normal method ChildClass’dan çağırılır. Fakat çağrılan method static ise early binding’de işlem yapılacağından çağrılan static method BaseClass’dan çağrılır. İşte bu özellikle method hiding, overriding’ten farklılaşmış olur .

Başlıksız notlar

* Maplerde key olarak immutable kullan, keySet gibi fonksiyonlarla patlama
* Kalıtım alındığında sorun yaşanılmaması için daima no args constructor yaz

Defensive Copies

* İmmutable bir objeye, mutable bir tip gönderiliyorsa defensive copy yapılmalıdır.
* New anahtar sözcüğü ile constructor çağırılarak yapılır.
* Amaç nesnenin tüm içeriğini değil, kendini korumaktır.
* Getterlarda ve constructorlarda kullanılabilir.

Örneğin, kişinin çocuklarını tutan kids adında bir array olsun. getKids fonksiyonu da bu arrayı döndürsün. Return kids yaparsak, dışarıdan kişinin çocuklarına müdahale edilebilir çünkü referans döndürüyoruz, bu yüzden return new ArrayList<>(kids) gibi bir dönüş yapılmalıdır. Constructorlarda da gönderilen parametrelerle sonradan nesne üzerinde değişiklik yapılmaması için gönderilen nesneler “this.start = new Date(start.getTime());” gibi bir kullanımla kullanılmalıdır.

Unmodifiable Collections

* Değiştirilemeyen koleksiyonlar (unmodifiable collections), değiştirme işlemlerini (add, remove, clear, replace, sort gibi) desteklemeyen koleksiyonlardır. Bu tür koleksiyonlar, genellikle başka bir koleksiyonun salt okunur görünümleri (wrappers) olarak kullanılır. Yani, değiştirilemeyen bir koleksiyonun içeriğini doğrudan değiştiremezsiniz, ancak temel koleksiyon hala değişebilir
* Değiştirilebilir olduğu içinde **immutable’**dan ayrışır.

Örneğin

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

Burada List.copyOf()’dan unmodifiable.Lİst döner ve studentsSecondCopy değişkeni unmodifiable olur. add, set, clear gibi methodlar error verir fakat son satırdaki bobsNotes’a yapılan işlem hata vermez. studentsSecondCopy içerisindeki bob nesnesinin, notes değişkenine verilen yazı eklenir.

* Mutator methodlar UnsupportedOperationException fırlatır.
* metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

  Açıklama otomatik olarak oluşturuldu

Instance initializer

Nesne yaratılırken constructordan önce çalışan kod bloğudur. Classda kaç tane olduğunun bir önemi yoktur, sırayla yukarıdan aşağıya çalıştırılırlar. Tüm instance initializerlar çalıştıktan sonra constructor çalışır.

* Bir sınıfın farklı constructor'ları olsa da, her constructor'da aynı başlangıç işlemleri gerekiyorsa, instance initializer bu işlemi constructor'lar arasında paylaşmak için kullanılabilir.
* Final değişkenleri initialize etmek için kullanılabilirler.

Static initializer

Classdan ilk defa nesne yaratılırken yani class oluşturulurken çalıştırılır. Çalışma mantığı instance initializer ile aynıdır. İlk olarak tüm static initializerlar çalışır, ardından instance initializerlar ve daha sonra constructor çalıştırılır. Data veya log bilgileri için kullanılabilir.

* Eğer sınıf yüklendiğinde bir veritabanı bağlantısı veya diğer dış kaynaklar açılması gerekiyorsa, bu tür başlangıç işlemleri için static initializer kullanabilirsiniz. Böylece sınıfın statik metotları veya sınıf üyeleri bu kaynakları kullanmaya başladığında hazır olurlar.
* Sınıfın statik üyeleri veya statik metotları için hesaplama işlemleri yapılması gerekiyorsa, static initializer kullanarak bu hesaplamaları yapılabilir.

***Initializerlar attributeları kullanabileceğinden tüm attribute tanımlarından sonra initializerlar eklenmelidir.***

Record Constructor

* **Canonical Constructor:** tüm attributelerin bulunması gerekir. Diğer constructorlar bunu kullanarak implemente edilebilir. **Long constructor** da denir.
* **Compact Constructor:** yalnızca recordlarda kullanılabilir. Conanical’ı kısa bir şekilde kullanma yoludur. Genelde bu constructorda canonical constructora verilen parametreler üzerinde kontrol, düzenleme gibi işlemler yapılır. Daha sonra bu constructor otomatik olarak kendi oluşturduğu conanical constructoru çağırır.

Enum Constructor

* Bir sınıf initialize edildiğinde, her enum değeri bir constructor çağırır. Çünkü değerler static final değerlerdir ve class initialize edilirken bir kere oluşturulurlar.
* Eğer bir constructor tanımlanmamışsa, bir default constructor oluşturulur.Ancak, eğer parametreli bir constructor mevcutsa, parametresiz bir constructora ihtiyaç vardır. Bu durumu ayrıca değerlerin yanına parametreleri girerek de aşabiliriz.
* Ek olarak, süslü parantezler {} kullanarak ek metodlar (toString gibi) veya benzer şeyler sağlayabiliriz.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

Final Classes

* Miras alınamayan classlardır.
* metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

  Açıklama otomatik olarak oluşturulduEnumlar ve Recordlar final classlardır.

Sealed Classes

* Bu özellik sınıfların miras verilmesini ve hangi alt sınıfların kullanılabileceğini denetlemeyi amaçlar.
* Sınıf tanımlamasında public key’inden sonra konulur.
* Permits keyword’ü ile hangi sınıfların bu sınıftan miras alabileceğini belirtiriz.

metin, ekran görüntüsü, diyagram, çizgi içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

Streams

* Orijinal veri yapısını değiştirmeden, bir aldığı girdi üzerinde işlem yapıp sonuç sağlar.
* Functional Programming yaklaşımını destekler.
* Chaining yöntemiyle tanımlama yapılabildiği için okunaklı ve kolay kod yazılabilir.
* Genel olarak **lazy** işlemler olarak çalışırlar. Yani stream işlemi yalnızca sonuç talep edildiğinde (terminal operation) gerçekleşir.
* Büyük verilerle uğraşırken parallel streams ile performans avantajı bulunur.
* Stream pipeline’ı stream ile başlar “bingoPool.stream(). …”, stream ve terminal operation arasındaki işleme “intermediate operation” denir.
* Sonuç her durumda tutarlı olsada işlemler her durumda tutarlı olmayabilir. Örneğin optimizasyon için intermediate operation’ların sırası değişebilir, bazıları birleştirilebilir veya atlanabilir. Bu otomatik olarak yapılır. Bu iyi bir şey olsada intermediate operation’ların yan etkileri varsa sorunlar ortaya çıkabilir bu yüzden yan etkisi olabilecek şeylerden kaçınmak gerekir.
* Terminal operation bir defa çağırıldıktan sonra tekrar çağırılamaz.
* stream'ler iki temel türde olabilir: **finite** ve **infinite**. Bu terimler, bir stream'in eleman sayısının sınırlı mı yoksa sınırsız mı olduğunu belirtir.
* **Finite Stream**: Bir sonlu stream, belirli bir sayıda elemana sahip olan ve elemanları sınırlı bir veri kümesinden alan bir stream'dir. Örneğin, bir liste veya diziyi temsil eden bir stream sonlu bir stream'dir çünkü eleman sayısı bellidir ve sınırlıdır.
* **Infinite Stream**: Bir sonsuz stream, eleman sayısı sınırsız olan ve gerektiğinde üretilebilen bir stream'dir. Örneğin, IntStream.iterate veya Stream.generate gibi yöntemlerle oluşturulan bir stream sonsuz bir stream'dir. Sonsuz stream'lerde elemanlar belirli bir koşula göre üretildiğinden, eleman sayısı sınırsızdır. Ancak bu tür stream'leri kullanırken dikkatli olmanız gerekebilir, çünkü sonsuz eleman sayısı bellek veya işlemci kaynaklarını tüketebilir.
* Stream Sources

metin, ekran görüntüsü, sayı, numara, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

dropWhile methodu predicate true olduğu sürece elemanları stream'den atıyor, false olduğu anda da atma işlemini bırakıyor, takeWhile ise true olduğu sürece elemanları streame alıyor. dropWhile başta dolu streamle başlayıp ondan çıkarmalar yaparken, takeWhile boş bir stream ile başlayıp içini dolduruyor.

* metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

  Açıklama otomatik olarak oluşturuldu

Peek’in amacı stream elemanlarına herhangi bir side effect’i olmamaktır.

* Stream’ı sonlandıran işlemlere **terminal operation** denir. forEach, count, min, max bunlara örnek.
* **Reduction Operation:** bir tür terminal işlemdir. Bu tür işlemler, bir koleksiyonun veya veri akışının elemanlarını birleştirerek daha basit bir sonuca dönüştürmek amacıyla kullanılır. Reduction işlemleri genellikle toplama, çıkarma, çarpma gibi matematiksel işlemleri ifade eder, ancak farklı veri türleri ve işlemler için de kullanılabilirler. Collect, reduce, toArray, toList reduction’a örnektir.
* **Collectors**: Java Stream API'sinin bir parçası olan ve Stream elemanlarını toplamak, gruplamak veya dönüştürmek için kullanılan bir yardımcı sınıf koleksiyonudur. Collectors sınıfı, Stream işlemlerinin sonucunu çeşitli biçimlerde toplamak ve işlemek için kullanılan yöntemler sağlar. Bu şekilde, Stream elemanlarını farklı koleksiyon tiplerine, haritalara veya özel sonuçlara dönüştürebilirsiniz.
* metin, yazı tipi, çizgi, sayı, numara içeren bir resim

  Açıklama otomatik olarak oluşturuldu

**Collect:** fonksiyonu, bir Stream'deki öğeleri bir koleksiyon veya başka bir sonuç yapısına toplamak için kullanılır. Genellikle bu fonksiyon, bir sonuç koleksiyonu oluştururken, her öğeyi işler ve sonucu toplar. Bu, Stream öğelerini gruplama, filtreleme veya dönüştürme gibi çeşitli senaryolarda kullanışlıdır.

**Reduce:** fonksiyonu, Stream içindeki öğeleri birleştirerek tek bir sonuç üretmek için kullanılır. Bu, toplama, çarpma, birleştirme gibi işlemler için uygundur. İki parametre alır: bir başlangıç değeri ve bir BinaryOperator işlevi. İşlem, her öğe üzerinde bu işlevi uygulayarak bir sonraki öğe ile birleştirir.

Optional Class

* Java 8'de tanıtılan ve özellikle null değerlerle ilgili sorunları ele almak için kullanılan bir sınıftır. Bu sınıf, bir değerin olup olmadığını temsil etmek veya bir metotun bir değer döndürebileceğini ancak bu değerin null da olabileceğini belirtmek için kullanılır.
* Optional döndüren bir fonksiyon hiçbir durumda null döndürmemelidir. Bunun yerine Optional.empty() döndürmelidir.
* Nesneleri optionala koymak memory harcar ve yürütme hızını yavaşlatır, okunabilirliği azaltır ve compexity’i artırır. Bu yüzden her yerde kullanmak yerine doğru yerde kullanılmalıdırlar.
* Serializable değillerdir.
* Fields veya method parametresi olarak kullanılmaları tavsiye edilmez.
* *metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

  Açıklama otomatik olarak oluşturuldu*
* Stream’lerde Optional döndüren fonksiyonlar  
  metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

  Açıklama otomatik olarak oluşturuldu

Static İmport

* Normal import yapar gibi yapılır fakat importtan sonra static key’i eklenir.  
  metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

  Açıklama otomatik olarak oluşturuldu

FlatMap

* Stream pipelineda one-to-many dönüşümler yapılır. İntermediate operation’dur.

Java.Math

* Overflow oluşabilecek durumlarda kontrollu artırma yapmak için **Math.incrementExact()** kullanılır. Overflow olduğunda exception fırlatır. Farklı exact fonksiyonları da bulunur.
* Math.abs() fonksiyonu min değerde pozitifi veremez, çünkü yoktur. Bu yüzden kullanırken dikkatli olmak gerek. Eğer integer min değer gelme ihtimali varsa parametre olarak long kullanmak gerek.
* Float 6-9 digite kadar destekler bu yüzden min, max gibi karşılaştırma işlemlerinde kesinlik için double kullanmak daha doğrudur.

Random

* ints methodu ile IntStream döndürülebilir. Parametre girerek origin ve bound belirlenebilir.

metin, yazı tipi, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu  
limit kullanmak yerine üç parametre ile Random.ints(limit, origin, bound) şeklinde yapılabilirdi.

* Değişken oluştururken seed kullanılabilir. Algoritmalar aynı seed değeri ile aynı sayıları üretirler. Bu debug, test, hata ayıklama gibi durumlarda hata oluşan senaryoyu tekrarlamak için kullanışlıdır. Ek olarak farklı sayılar elde etmek içinde farklı seedler verilebilir.

Big Decimal

* Bu sınıf, kesirli sayıları ve işlem sonuçlarını daha yüksek hassasiyetle işlemek için tasarlanmıştır. Genellikle finansal hesaplamalar, vergi hesaplamaları, para birimi dönüşümleri gibi durumlarda kullanılır.
* Değer binary taban yerine iki adet Integer alanda tutulur.
* Hesaplamalarda hassasiyeti kaybetmeden sayıların nasıl yuvarlanacağını kontrol etmemizi sağlar.
* Constructorda double değerler kullanmamak gerekir. valueOf methodu da dahil.
* metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

  Açıklama otomatik olarak oluşturuldu
* setScale kullanıldığında yuvarlamanın nasıl yapılacağı belirtilmelidir, RoundingMode enumıyla parametre olarak gönderilmelidir.

Immutable bir class olduğundan setScale’den dönen nesneyi bd’e atadık.

* Float, Double gibi wrapperlarda kullanılan \_, F, L gibi ekler BigDecimal constructorunda parametre olarak String verildiğinde gönderilemezler.
* İçerisinde, sık kullanılan sayılar static BigDecimal nesnesi olarak bulunurlar. One, Zero, Ten.

Java.time

* ***OffSetDateTime*** ve ***ZonedDataTime*** sınıfları tarih ve saait evrensel standartlara göre ölçmek ve saklamak gerektiğinde kullanılır.
* Java.time.temporal paketi java’nın bazı önemli arayüzlerini içerir.
  + ***Temporal*** ve ***TemporalAccessor*** bir tarih saat nesnesinden okumanın veya bu nesneye yazmanın tek tip bir yolunu tanımlarlar.
  + ***TemporalAdjuster***, ***TemporalAmount***, ***TemporalField***, ***TemporalUnit*** bir tarih saat nesnesi hakkında ne tür bilgiler istediğinizi belirtmek için method parametresi olarak sıklıkla kullanılır.
* ***TemporalAdjusters*** yılın ilk günü, son günü, gelecek ay, gelecek yıl gibi yararlı tarihler verebilen methodlarıyla ***TemporalAdjuster*** döndüren bir helper sınıftır.
* May5 bir LocalDate nesnesidir.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

* Plus, minus gibi methodlar da with methodları gibidir. Tarihe ekleme çıkarma yapabiliriz.
* metin, yazı tipi, ekran görüntüsü, grafik içeren bir resim

  Açıklama otomatik olarak oluşturuldu

*datesUntil stream döndürür*.

* ***Instant*** class’ında saniye ve nanosaniye tutulur. Yalnızca zaman içinde bir noktayı veya bir zaman damgasını temsil etmek üzere tasarlanmıştır.
* GMT ve UTC arasında günde 0.9 saniye fark oluşur.
* DateFormat, SimpleDateFormat, Date, TimeZone, GregorianCalender sınıfları JDK8’den önce kullanılmaktadır. Artık immutable thread-safe classlar (java.time) kullanılıyor ve kullanılması öneriliyor.
* ***ZoneId*** tarih ve saat değerlerini farklı coğrafi bölgelere göre ayarlamak ve dönüştürmek amacıyla kullanılır.

ResourceBundle

* bir uygulamanın farklı dillerdeki metinleri ve diğer kaynakları yönetmek için kullanılan bir sınıf ve mekanizmadır.
* ResourceBundle sınıfı, metinleri ve diğer verileri anahtar-değer çiftleri olarak saklar. Her bir anahtar, bir dildeki metni veya kaynağı temsil eder. Farklı diller için ayrı ayrı ResourceBundle dosyaları oluşturulur. Bu dosyalar, genellikle .properties uzantılı dosyalardır ve anahtar-değer çiftlerini içerir.
* Resource Bundle plugin’i ile daha kolay kontrol edilebilirler.

Regular Expressions

* Metin işleme ve arama işlemlerinde kullanılan güçlü bir desen eşleme aracıdır. Düzenli ifadeler, metin içinde belirli desenleri tanımlamak, bulmak veya değiştirmek için kullanılır.
* Desen eşleme, metin doğrulama, metin düzenleme, veri ayıklama gibi durumlarda kullanılabilirler.
* Java.util.regex paketinde yerleşik olarak regex classları bulunmaktadır. Bu classlar Pattern ve Matcher classlarıdır.

Kurallar

* + ^, -, ], \ karakterleri dışındaki herhangi bir karakter köşeli parantez içerisindeyse örneğin [a], bu gerçekten o karakteri belirtiğimiz anlamına gelir, bir meta karakter değildir.
  + metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

    Açıklama otomatik olarak oluşturuldu
  + metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

    Açıklama otomatik olarak oluşturuldu
  + metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

    Açıklama otomatik olarak oluşturuldu  
    [Pattern (Java SE 17 & JDK 17) (oracle.com)](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/regex/Pattern.html)

Pattern ve Matcher Sınıfları

* Regex kabul eden bir fonksiyon kullanıldığında (örneğin String.matches()), verilen string Pattern sınıfı üzerinden regex olarak işlem görür.
* **Pattern** sınıfının kullanımı, regular expressions’ları işlemek ve metin üzerinde eşleşmeler bulmak için birçok avantaj sunar:
  + **Performans İyileştirmesi:** Regular expression’lari derleyerek Pattern nesnesi oluşturmak, aynı regular expression’ı birden fazla kez kullanmanın daha hızlı olmasını sağlar. Derleme işlemi, regular expression’ı daha verimli bir şekilde işlenen bir ara formata dönüştürür.
  + **İfade Okunabilirliği:** Derlenmiş bir Pattern nesnesi, kodunuzun daha okunabilir olmasını sağlar. Tek bir Pattern nesnesi, regular expression’in belirli bir yerde derlendiğini ve ne amaçla kullanıldığını açıkça belirtir.
* **Matcher** sınıfı, Java'nın , regular expression işleyicisi içinde yer alır ve Pattern nesneleri ile birlikte kullanılarak metin üzerinde eşleşmeleri bulmanıza olanak tanır. Matcher sınıfının kullanımı, metin işleme süreçlerinde bir dizi avantaj sunar:
  + **Metin İşleme ve Eşleşme Bulma:** Matcher sınıfı, belirli bir regular expression’a göre metin içinde eşleşmeleri bulmanıza yardımcı olur. Bu, metin analizi, veri çıkarma veya örüntü arama gibi birçok senaryoda kullanışlıdır.
  + **Çoklu Eşleşme İşlemleri:** Matcher nesnesi, metin içinde birden fazla eşleşme bulma yeteneği sunar. Bu sayede metindeki tüm eşleşmeleri tespit edebilir ve işleyebilirsiniz.
  + **Eşleşme Bilgileri:** Matcher sınıfı, eşleşen metin parçalarının başlangıç ve bitiş konumlarını, uzunluklarını vb. gibi ayrıntıları elde etmenize olanak tanır.
  + **Esneklik ve Özelleştirme:** Matcher nesnesi, metin eşleştirmesi sırasında farklı ayarlar ve seçenekler belirlemenize olanak tanır. Bu sayede eşleşme işlemi davranışını özelleştirebilirsiniz.
* **Matcher** sınıfının bir örneği, üzerinde işlem yapıldıkça değişen bir duruma sahiptir. Yani matcher ***thread safe değildir.***
* **Greedy** ve **Reluctant** regular expressionslar vardır.
  + **Greedy:** eşleşebildiği kadar karakterle eşleşmeye çalışır. Örneğin ‘.\*’ eşleşebildiği kadar karakterle eşleşmeye çalışır.
  + **Reluctant** ise olabildiğince az karakterle eşleşmeye çalışır.
* Regular expression’lar default olarak **greedy**’dirler. Fakat ‘**?**’ quantifier modifier ile **reluctant** yapılabilirler.  
  Örneğin  
  **"I like B.M.W. motorcycles."** Cümlesi için  
  **[A-Z].\*[.]** bu regex ile büyük harfle başla sonradan herhangi sayıda herhangi karakter gelsin ve nokta ile bitsin diyoruz. ***Greedy regextir.*** Sonuç : **"I like B.M.W. motorcycles."  
  [A-Z].\*?[.]** ise büyük harfle başla sonradan herhangi sayıda herhangi karakter gelsin ama ilk noktada bitsin diyoruz. ***Reluctant regextir.*** Sonuç : **"I like B."**
* Matcher.find() methodu son kalınan yerden devam eder. Bu yüzden eğer baştan başlamak istiyorsak Matcher.reset() ile resetlemek gerekir.
* metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

  Açıklama otomatik olarak oluşturuldu

Bu fonksiyonlar kullanıldığında, Matcher sınıfının içerisinde bulunan groups dizisinin içi başlangıç ve bitiş indeksleriyle doldurulur. group methodu ile bulunan değer String olarak döndürülebilir. reset fonksiyonundan sonra groups sıfırlanır.  
  
group methodunun asıl güçlü olduğu nokta regex içerisinde gruplandırma yapabiliyor oluşumuzdur.  
Örneğin  
**"<[hH]\\d>.\*</[hH]\\d>**" şeklinde bir regex’imiz olsun. Bu regex bir html headerını aramaya yarar. Yukarıdaki methodlardan biri çalıştığında bulunan değer group(0) ile elde edilebilir.

Regex’e parantezler ile istediğimiz alanları da belirtebiliriz.

**"<[hH](\\d)>(.\*)</[hH]\\d>**" regex’imiz böyle olsun.

**<H1>My Heading</H1>** String’imiz de bu şekilde.

**group(0)** = <H1>My Heading</H1>

**group(1)** = 1

**group(2)** = My Heading

şeklinde olur. Bu işleme **Capturing** denir.

* **Capturing’**ler “?<name>” şeklinde adlandırılabilir.  
    
  regex’imiz **"<[hH](?<Num>\\d)>(?<Content>.\*)</[hH]\\d>**" olsun.

Group fonksiyonuna isim verilerek erişilebilir.

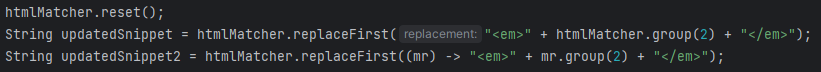
group(Num) = 1

group(Content) = My Heading

* İntellij da bir regex’in üzerinde durup soldaki ampüle tıklanıp “Check RegExp” ile regex üzerinde deneme yapılabilir.

metin, ekran görüntüsü, multimedya yazılımı, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

* 

İki replaceFirst methodu da aynı işi yapıyor gibi görünsede, ilk kod satırı hata verir. Çünkü htmlMatcher içerisindeki groups, reset fonksiyonu çağırıldığı için boştur. Parametre olarak group verilmeye çalışıldığında ise hata döndürülür.

Bu sorundan lambda expression ile kurtulunabilir. İlk kod satırında group methodu herhangi bir match methodu çağrılmadan çağrılır hata verme sebebi budur. Fakat ikinci kod satırında

Lambda expression olarak gönderildiğinden bu lambda expression’da replaceFirst methodunun içerisinde match methodu olan find() çağrıldıktan sonra çağrıldığı için find() groups’u doldurduğundan hata vermez ve çalışır.

Bu işlem **back reference** ile de yapılabilir.

Back Reference

* Bir capturing grubunu tekrar etmenin bir yoludur.

**"<([hH]** **\\d)>(.\*)</[hH]\\d>**" regex’imiz böyle olsun.

**"<([hH]\\d)>(.\*)</\\1>**" ifadesindeki “\\1” kısmı 1.capturingi gösteriyor.

* Tekrar eden desenleri yazmayı ve okunmasını kolaylaştırır.
* Javada replacement String’de (Fonksiyonlarda parametre ismi String replacement şeklindedir) “$” işareti ile başlar. “$” işaretinden sonraki sayı fonksiyon içerisinde group(sayı) şeklinde group fonksiyonu çağrılıyormuş gibi davranır ve o yere group fonksiyonundan dönen değer yazılır. (Fonksiyon içerisinde group fonksiyonu çağrılmaz, sadece verdikleri sonuç aynıdır.)

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

Naming Convention

* **Packages**
  + Daima lower case olurlar.
  + Unique olmalılar.
  + Reversed internet domain name kullan, “com.meteahmetyakar.mylibrary” gibi. Domainde ‘-‘ varsa ‘\_’ olarak yaz. sayı varsa, java keyword varsa başına ‘\_’ ekle.

1world.com 🡪 com.\_1world  
Switch.supplier.com 🡪 com.supplier.\_switch  
Experts-exchange.com 🡪 com.experts\_exchange

* **Class**
  + CamelCase olurlar
  + İsim olmalıdırlar
* **Interface**
  + CamelCase
  + Interface’i uygulayacakların ne olduğunu veya ne yapabileceklerine göre isimlendir.  
    List, Comparable, Serializeable gibi.
* **Method**
  + mixedCase.
  + genelde fiil.
  + ne yaptığını veya ne döndürdüğünü belirtir.

size(), getName()

* **Constants**
  + UPPERCASE
  + Kelimeler ‘\_’ ile bölünür.

MAX\_INT, SEVERITY\_ERROR

* **Variable** and **Fields**
  + mixedCase
* **Type Parameters**
  + Tek ve büyük Karakter olur.
  + E (Element), K (Key), T (Type), V (Value), S, U, V etc – 2nd, 3rd, 4th types

Java IO, NIO ve NIO.2

* **Java IO (Input/Output):** Java IO, temel giriş ve çıkış işlemlerini gerçekleştirmek için kullanılan klasik bir I/O API'sidir. java.io paketi altında bulunur ve karakter tabanlı ve bayt tabanlı verilerle çalışabilir. Bu API, InputStream ve OutputStream gibi sınıfları içerir ve dosya okuma/yazma, soketlerle iletişim kurma ve veri akışı işlemlerini yönetmek için kullanılır.
* **Java NIO (New I/O):** Java NIO, Java 1.4 sürümü ile tanıtılan bir I/O API'sidir. Bu API, java.nio paketi altında bulunur. Java NIO, gelişmiş I/O işlemleri için tasarlanmıştır ve kanallar **(channels)** ve seçiciler **(selectors)** gibi yeni kavramlar içerir. Bu sayede daha verimli ve hızlı I/O işlemleri gerçekleştirmek mümkün olur. Java NIO, özellikle ağ programlaması ve büyük veri işleme uygulamaları için uygundur.
* **Java NIO.2 (New I/O 2.0):** Java NIO.2, Java 7 sürümü ile birlikte tanıtılan bir güncellemedir. Bu API, java.nio.file paketi altında bulunur ve dosya sistemi işlemleri için kullanılır. Java NIO.2, Java'nın önceki sürümlerine göre daha zengin bir dosya işleme yeteneği sunar. Dosya oluşturma, silme, taşıma ve dosya özelliklerini sorgulama gibi işlemleri daha kolay hale getirir. Ayrıca, sembolik bağlantılar ve dosya sistemi gözlemleme (**watch service**) gibi özellikleri de içerir.

*Hangi API'nin kullanılacağı, projenizin gereksinimlerine ve performans beklentilerine bağlıdır. Basit dosya işlemleri için Java IO yeterli olabilirken, daha karmaşık veya performans odaklı işlemler için Java NIO veya NIO.2 tercih edilebilir.*

A screenshot of a computer program

Description automatically generated

Exceptions

**Throwable**

* Java'daki hata ve istisna mekanizmasının temelidir ve tüm hata ve istisna sınıfları için üst sınıf olarak kullanılır.
* Throwable sınıfının iki ana alt sınıfı vardır:

A diagram of a computer

Description automatically generated**Error:** Bu alt sınıf, genellikle sistemle ilgili kritik hataları temsil eder ve programın normal olarak kurtarılamayacak bir duruma düştüğü durumları ifade eder. Örnek olarak, "OutOfMemoryError" veya "StackOverflowError" gibi hatalar bu kategoriye girer.

**Exception:** Bu alt sınıf, programın çalışma zamanında ortaya çıkan ve program tarafından yakalanabilen istisnaları temsil eder. Exception sınıfı, daha önce tanımlanan özel istisna sınıflarının ve kullanıcı tarafından oluşturulan istisna sınıflarının temel sınıfıdır. Özel bir istisna türü tanımlamak için Exception sınıfından türetilebilirsiniz.

* Javada **Checked exception** ve **unchecked** **exception** olarak iki tür exception vardır.

**Checked Exception**

* Meydana gelebilecek beklenen veya yangın bir sorunu temsil eder. Örneğin açılacak dosyanın bulunamaması gibi.
* Bir fonksiyon veya constructor tanımlarken imzasına **throws “ExceptionName”** yazılarak oluşturulur. Bu fonksiyon veya constructor kullanılırken exception’un handle edilmesi gerekir. Aksi takdirde compile edilirken hata verir.
* Örneğin bir oyun yazıyoruz ve dosyadan verileri okuyacağız, okuma fonksiyonunda dosyayı açmaya çalıştık ve açamadık. Burada try-catch ile yakalama yapmak yerine bu istisnayı fırlatmak daha mantıklıdır çünkü veriler okunmazsa zaten oyun çalışmayacaktır yani dosya okunamadığında catch içerisinde yapılacak çok bir şey de yoktur.

**Unchecked Exception**

* Bu tür istisnalar runtime’da kontrol edilir. Program çalışırken ortaya çıkan hatalardır. Örneğin bir array’in sınırları dışına çıkılması veya bir sayıyı sıfıra bölmek gibi. Bu tür istisnaları try-catch ile ele almak yerine if ile hata önleme (prevention) önlemleri almak gerekir. İf ile null kontrolu gibi.

**Suppressed Exceptions**

* Fırlatılan fakat daha sonra bir şekilde görmezden gelinen exceptionlara denir. Örneğin try-with-resources kullandığımızı varsayalım ve try bloğunda bir exception fırlatılsın daha sonra try bloğundan exception ile çıkılırken verilen kaynaklar kapatılırken de bir exception fırlatılsın. Bu durumda ilk exception ikinci exceptionu bastırır ve catch bloğunda try bloğundan fırlatılan istisna yakalanılır, kapatma işlemindeki istisna es geçilir.
* Suppressed exceptionları ele almak için getSuppressed() methodu kullanılabilir.
* addSuppressed() ile de bir exception’un içine suppressed olması için başka bir exception eklenebilir.

metin, yazı tipi, ekran görüntüsü, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

Try-Catch-Finally

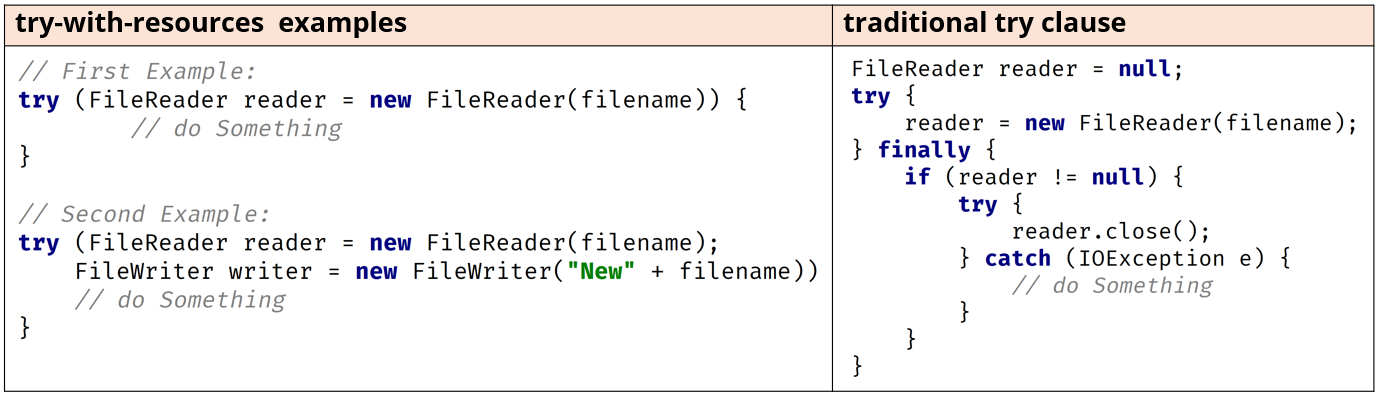
* | işareti ile çoklu hata yakalanılabilir.

**Finally Block**

* Finally bloğu her koşulda çalıştırılır yani return, break, continue, throw gibi anahtar kelimelerle bypass edilemez. Örneğin bir fonksiyon içerisinde try-catch kullanıyor ve return ile değer döndürüyorsanız. Finally bloğu return’den sonra olsa bile çalıştırılır.
* özellikle kaynakların (dosya işlemleri, veritabanı bağlantıları, ağ bağlantıları vb.) serbest bırakılması veya kapatılması gereken durumlarda kullanılır. Bu, kaynakların sızıntı yapmasını önler.

**Try-With-Resources**

* Bu özellik, kaynakları (dosya işlemleri, veritabanı bağlantıları, ağ bağlantıları vb.) açma ve otomatik olarak kapatma işlemlerini kolaylaştırır. Ayrıca, kaynakların güvenli bir şekilde kapatılmasını ve kaynak sızıntılarını önlemeyi sağlar.
* try bloğu içinde açılan kaynaklar try-with-resources ifadesinin parantezlerinde tanımlanır. *Bu kaynaklar,* ***AutoCloseable*** *arabirimini uygulayan sınıflardan olmalıdır.*
* Kaynaklar otomatik olarak kapatılır. try bloğu tamamlandığında veya hata durumunda, kaynaklar close() metodu çağrılarak otomatik olarak kapatılır. Bu, kaynakların sızıntı yapmasını önler.
* Kaynakları açarken birden fazla kaynağı aynı anda açabilirsiniz. Bu, kodunuzu daha okunaklı hale getirebilir.
* Kaynakları kapatma işlemi gizlidir ve kullanıcı tarafından doğrudan yapılmaz. Ancak, kaynakların otomatik olarak kapatılmasını sağlamak için close() metodunu kendi kaynak sınıflarınızda uygulamanız gerekir.



* Özellikle Files class’ında stream döndüren herhangi bir method kullanılıyorsa try-with-resources kullanmak çok önemlidir.

**LBYL ve EAFP**

* **LBYL (Look Before You Leap):** bir işlemi yapmadan önce koşulları kontrol etmeyi ve işlemin güvenli olduğundan emin olmayı önerir.
* **EAFP (Easier to Ask for Forgiveness than Permission):** önce bir işlemi yapmayı dener ve hata oluşursa bu hatayı işler. Java'da ise bu tip bir yaklaşım, try-catch blokları ile uygulanabilir.
* Hangisinin kullanılacağı duruma göre seçilir. Örneğin kullanıcıdan yalnızca harfler içeren bir girdi alınacaksa **LBYL** yapılırsa tüm harfleri kontrol etmek gerekir ama **EAFP** ile hata alındığında yanlış girdi alındığı anlaşılabilir. Böylece daha performanslı bir kod yazılmış olur. Ya da bir key’in map’de olup olmadığını kontrol etmek için **LBYL** kullanmak daha mantıklıdır.
* Güvenlik önemliyse ve hatalar ciddi sonuçlar doğurabilirse, **LBYL** tercih edilebilir.
* metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

  Açıklama otomatik olarak oluşturulduPerformans kritik bir faktörse veya hatalar beklenmedik türdense **EAFP** daha uygun olabilir.

Working With Files

**File Handle**

* Bir dosyaya işaret eden ve işletim sistemi tarafından kullanılan bir referanstır. Dosyanın kendisi değil, dosyanın işletim sistemi tarafından takip edilmesini sağlayan soyut bir temsilidir. Dosya işleyici, dosyanın konumunu ve durumunu işaret eder, ancak dosyanın kendisi verileri içermez.

**File Resource**

* Bir dosyanın kendisi tarafından depolanan gerçek verileri ifade eder. Bu veriler, disk üzerinde saklanır ve işletim sistemi ve uygulamalar tarafından erişilebilir. Dosya kaynağı, dosyanın içeriğini temsil eder.

Özetle, ***File Handle***, dosyanın soyut bir temsilidir ve işletim sistemi tarafından dosyayı izlemek ve yönetmek için kullanılırken, ***File Resource***, dosyanın kendisinde depolanan gerçek verileri ifade eder. ***File Handle***, dosya kaynağına erişim sağlayan bir araç olarak kullanılır. Bu ayrım, dosya işleme ve yönetme konularında önemlidir, çünkü dosyanın kendisi ile dosyanın işlenmesi ve kullanılması arasındaki farkı açıklar.

**File Class**

* **java.io.File** sınıfı, dosyaları ve dizinleri temsil etmek ve dosya işlemleri gerçekleştirmek için kullanılır.
* Temel dosya işlemleri için kullanılır, yani dosya veya dizin oluşturma, silme, yeniden adlandırma, kontrol etme, dosya yolu alma ve benzeri işlemleri yapabilir.

**Files Class**

* **java.nio.file.Files** sınıfı, Java 7 ve sonraki sürümlerde tanıtılan NIO (New I/O) paketinin bir parçasıdır ve modern dosya işleme işlevselliği sunar.
* Files sınıfı, dosya veya dizinlerle ilgili **File** a göre daha gelişmiş işlemler gerçekleştirmenizi sağlar. Örneğin, dosya kopyalama, taşıma, symbolic links oluşturma ve açma, dosya içeriği okuma ve yazma gibi işlemler için kullanılabilir.
* Files sınıfı, Path nesneleri ile çalışır ve daha fazla işlemi daha basit ve güçlü bir şekilde gerçekleştirmenizi sağlar.
* Asynchronous dosya I/O işlemleri.
* Bir dosyanın tamamını kilitlemek yerini belli bölgelerini kilitleyebilir.
* Multiple threads destekler.
* FileChannel ile dosyaları doğrudan bellekten tamponlara okuyarak ve yazarak belleği daha etkili kullanır ve daha verimli bir şekilde yönetir
* **metin, ekran görüntüsü, sayı, numara, yazı tipi içeren bir resim

  Açıklama otomatik olarak oluşturuldugetAttribute()** veya **readAttributes()** methodları ile dosyanın bir çok özelliğine erişilebilir.
* **Files.list()** methodu ile path’deki dosya ve dizinlerin listesini alabiliriz. Ama dosya ve dizinlerin içerisindeki dosya ve dizinlere erişemeyiz.
* **Files.walk()** methodu ile **list()** methodunun yaptığının aynısı yapılır fakat maxDepth parametresi ile dosya ve dizinler içerisindeki dosya ve dizinlere de erişilebilir.
* **Files.find()** methodu ile path içerisinde arama yapılabilir. Derinlik de belirtilebilir.
* **Files.newDirectoryStream()** methodu ile dosya ve dizinlerin listeleyip DirectoryStream nesnesi olarak döndürülebilir. Çeşitli filtreler kullanarak belirli türdeki dosyaları veya dizinleri seçmenizi sağlar. Örneğin, sadece belirli bir uzantıya sahip dosyaları veya belirli bir özelliklere sahip dosyaları listelemek için bu yöntemi kullanabilirsiniz.  
    
  bu methodun **glob** adı verilen bir string kullanarak hangi yolları istediğimizi filtreleyebiliriz.  
    
  **glob,** regular expressionslara benzeyen fakat daha basit bir syntax’ı olan bir *“limited pattern language”* dir.

[glob patterns](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/nio/file/FileSystem.html#getPathMatcher(java.lang.String))  
  
**metin, ekran görüntüsü, yazılım, multimedya yazılımı içeren bir resim

Açıklama otomatik olarak oluşturuldu**

***glob yerine lambda expression kullanılarak daha spesifik filtrelemeler de yapılabilir.***

* **Files.walkFileTree()** methodu dosya veya dizin ağaçları üzerinde gezinmek ve işlem yapmak için kullanışlıdır.  
  + Belirli bir dizin ve alt dizinlerini tarayarak dosya veya dizinlerin listesini almak.
  + Her dosya veya dizin üzerinde belirli bir işlem gerçekleştirmek (örneğin, dosyaları okumak veya silmek).
  + Dosya veya dizinleri belirli kriterlere göre filtrelemek (örneğin, belirli bir uzantıya sahip dosyaları bulmak).
  + Bu method ile mevcut klasörler üzerinde gezinti yapılabilabilir. Örneğin “public/assets/icons” şeklinde bir path’iniz var, walkFileTree ile bu pathde gezerken klasör yoksa oluşturmak istiyorsunuz. (Public yoksa oluştur, içine gir.) Böyle bir şey **yapılamaz**.

**Files.walkFileTree** methodu, bir **FileVisitor** nesnesi kullanarak gerçekleştirilir. **FileVisitor**, dosya ve dizinleri işlemek için kullanılan bir arayüzdür ve aşağıdaki metodları içerir:

**FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs):** Bir dizini ziyaret etmeden önce çağrılır.

**FileVisitResult visitFile(Path file, BasicFileAttributes attrs):** Bir dosyayı ziyaret ettiğinizde çağrılır.

**FileVisitResult visitFileFailed(Path file, IOException exc):** Bir dosya ziyaret edilirken hata oluştuğunda çağrılır.

**FileVisitResult postVisitDirectory(Path dir, IOException exc):** Bir dizini ziyaret ettikten sonra çağrılır.

Bu metodlar, Files.walkFileTree yöntemi tarafından otomatik olarak çağrılır ve kullanıcının belirlediği işlemleri gerçekleştirmek için uygulanabilir. SimpleFileVisitor sınıfı kalıtım alınarak veya Anonymous class olarak implemente edilerek bu metodlar override edilebilir ve isteğe göre şekillendirilebilir.

* **Files.move()** methodu ile dosya taşıma veya yeniden adlandırma işlemleri yapılabilir. Eğer dosya yolları aynı sadece sondaki dosya ismi farklıysa yeniden adlandırma, dosya yolları farklıysa taşıma işlemi yapılır.

Input Streams

* Byte tabanlı verileri okumak için kullanılır ve bu verileri okuma işlemlerini soyut bir arayüz olarak temsil eder. Bu sınıf, özellikle dosya okuma, ağ üzerinden veri alımı, veri sıkıştırma ve benzeri işlemlerde kullanılır.
* InputStream soyut sınıfı, alt sınıflar tarafından uygulanır ve bu alt sınıflar çeşitli kaynaklardan veri okuma işlemlerini destekler. İşte bazı yaygın InputStream alt sınıfları:
* Byte stream veya character stream döndürürler.

**FileInputStream:** Bir dosyadan veri okumak için kullanılır. Read methodu sürekli Disk Read yaptığı için çok verimsizdir.

**ByteArrayInputStream:** Bir byte dizisinden veri okumak için kullanılır.

**BufferedInputStream:** Bir başka InputStream'ı tamponlayarak veri okuma işlemini daha verimli hale getirir.

**DataInputStream:** Temel veri türlerini (int, float, double, vb.) okumak için kullanılır.

**ObjectInputStream:** Nesneleri serileştirmek ve deserileştirmek için kullanılır.

**PipedInputStream:** İş parçacığı veya işlemler arasında veri iletmek için kullanılır.

Readers

* Karakter tabanlı veri okuma işlemlerini gerçekleştirmek için kullanılan bir soyut sınıftır. Reader, özellikle metin dosyalarını veya karakter tabanlı verileri okumak için kullanılır ve karakterlerin Unicode karakter kodlarına dayalı işlem yapar.
* Reader sınıfı, veriyi okumak için soyut bir arayüz sağlar ve alt sınıflar tarafından uygulanır. En yaygın kullanılan Reader alt sınıflarından bazıları şunlardır:

**FileReader:**

* Java'da dosya okuma işlemlerini gerçekleştirmek için kullanılan bir sınıftır. Bu sınıf, metin tabanlı dosyaları okumak için kullanılır ve Reader sınıfının alt sınıfıdır.
* read() yöntemi ile dosyanın içeriği karakter karakter okunur. read() yöntemi bir karakterin Unicode değerini döndürür veya dosyanın sonuna geldiyse -1 döndürür. Bir buffer verilerek her seferde o buffer da doldurulabilir. Böylece *Disk Read* azaldığı için maliyet de azalır.
* close() yöntemi ile FileReader nesnesi kapatılır. Dosya okuma işlemi tamamlandığında kaynakların serbest bırakılması önemlidir.
* txt, csv, xml, json gibi dosya türleri okunabilir. Ek olarak .properties gibi yapılandırma dosyaları da okunabilir.

**BufferedReader:** Başka bir Reader'ı tamponlayarak karakter okuma işlemini daha verimli hale getirir. FileReader’a göre daha büyük buffer boyutu vardır. Buffer size’i constructorda belirtilebilir fakat default değerin çoğu durum için yeterlidir. Metin satırlarını okumak için kolaylık sağlayan methodları da bulunmaktadır.

**StringReader:** Bir karakter dizisinden (String) veri okumak için kullanılır.

**InputStreamReader:** InputStream'dan gelen byte'ları karakterlere dönüştürmek için kullanılır.

**Reader.transferTo()**

* bir reader’dan başka bir reader’a veri aktarımı yapılır.  
    
  metin, ekran görüntüsü, yazılım, yazı tipi içeren bir resim

  Açıklama otomatik olarak oluşturuldu

Writers

**Files Class**

* **Files.write()** ve **Files.writeString()** methodları ile dosyaya yazma işlemi yapılabilir. Bu methodlar dosya açma, yazma ve kapama işlemlerini kendileri yaptıkları için açma kapama kontrolleri yapılması gerekmez. Bu methodlar eğer OpenOption verilmezse Default Open Optionları kullanırlar.  
    
  **metin, ekran görüntüsü, yazı tipi, çizgi içeren bir resim

  Açıklama otomatik olarak oluşturuldu**  
  Bu sebeplede OpenOption verilmeden çağrıldıklarında dosyanın üzerine yazarlar.

**BufferedWriter, FileWriter and PrintWriter**

**metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu**

* Classlar flush’ı kendileri yaparlar fakat istediğimizde fonksiyonunu çağırarak biz de yaptırabiliriz. *Thread’lerde flush edilmemiş veriler diğer Thread’ler tarafından görülemeyeceği unutulmamalı.*

RandomAccessFile Class

* Bu sınıf, dosyanın herhangi bir noktasına doğrudan erişim (rastgele erişim) sağlar. Diğer giriş/çıkış sınıfları gibi seri veya tamponlu giriş/çıkış işlemleri yapmaz, bunun yerine dosyanın belirli bir konumuna gitmenize ve orada okuma veya yazma işlemleri yapmanıza izin verir.
* Dosya sisteminde çok büyük bir byte array’i gibi davranır.
* ***File pointer*** adında erişimde kullanılan bir tür cursor’a sahiptir.
* Okuma ve yazma işlemleri yapılabilir.
* Özel methodlar kullanılarak binary Data okunabilir ve yazılabilir.
* Pointerla ramde geziyor gibi işlem yapılır. Örneğin Student nesnelerini tutan bir array olsun ve her Student nesnesi 50 bayt olsun, array üzerinde gezme işlemi yaparken currentPtr + 50bayt gibi bir yöntem kullanılmalıdır. Burada da aynısı geçerli.
* A diagram of a record

  Description automatically generatedEğer her veri aynı büyüklükte değilse? Bu durumda her verinin başlangıç noktasının yazılı olduğu farklı dosya tutarız. Ya da tek bir dosya içerisinde hem filepointer’lar hem de data’lar tutulur.

metin, ekran görüntüsü, yazı tipi, sayı, numara içeren bir resim

Açıklama otomatik olarak oluşturuldu

*File Pointer kısmı gezinme işlemi için kaydedilmesi gerekir.*

* **RandomAccessFile.seek()** methodu ile **filepointer**’ı istenilen noktaya atayabiliriz.

DataStreams

* Özellikle veriyi daha karmaşık veri türlerine (örneğin, Java nesnelerine) dönüştürmek (Serialization) veya bu tür veriyi serileştirilmiş veriden geri döndürmek (Deserialization) için kullanılır.

**DataOutputStream**

* Primitive ve String java veri türlerini taşınabilir bir şekilde bir output stream’e yazmasını sağlar.

**DataInputStream**

* Veri akışlarından veri okumak için kullanılır. Primitive ve String veri türlerini okuyabilir.

**ObjectOutputStream**

* Nesnelerin taşınabilir bir şekilde bir output stream’e yazmasını sağlar.

**ObjectInputStream**

* Veri akışlarından veri okumak için kullanılır. Obje okuyabilir.

Serialization

* Bir veri yapısını veya nesneyi bir dosyada saklanabilecek bir biçime dönüştürme işlemine denir. Yalnızca **Serializable** interfacesini implemente etmiş ***Serializable*** classlar serileştirilebilir.
* Default serialization mekanizması nesnenin sınıfını, sınıf imzasını ve statik olmayan alanların değerlerini yazar. Bunlar sayesinde nesne ve onun state’leri read işleminde tekrardan oluşturulabilir. Bu işleme de ***Deserialization*** denir.
* Serialization ve deserialization işlemlerinde bazı durumlarda Incompatible Change (değiştirme uyumsuzluğu) ortaya çıkar, java bunu **serialVersionUID** alanını kontrol ederek yapar. Uyumsuzlukların sebepleri şunlardır;
  + Bir primitve alanın tipini değiştirmek. Örneğin int bir alanı long’a çevirmek, ilk bakışta yanlış gibi görünmese ve bir long’un int’i içerebileceği mantıklı gelsede biri 4 byte diğeri 8 byte olduğu için değişiklik sonrası dosyadan okurken hata meydana gelir.
  + Field silmek.
  + Non-static alanı static yapmak veya non-transient alanı transient yapmak.
  + Bir sınıfı hiyerarşisi içinde taşımak, daha önce serileştirmek için kullandıktan sonra writeObject() ve readObject() methodlarını değiştirmek gibi daha karmaşık başka değiştirme uyumsuzlukları da vardır.
  + Tüm uyumsuzluklar için;  
    [Java Object Serialization Specification: 5 - Versioning of Serializable Objects (oracle.com)](https://docs.oracle.com/en/java/javase/17/docs/specs/serialization/version.html#incompatible-changes)
* Serialization ve deserialization işlemlerinde Compatible Change (Uyumlu Değişiklik) de bulunmaktadır. Bunlar;
  + Field eklemek.
  + writeObject() ve readObject() methodlarını eklemek.
  + Access modifier değişikliği.
  + Static bir değişkeni non-static yapmak veya transient alanı non-transient yapmak.
  + Tüm uyumlu değişiklikler için;  
    [Java Object Serialization Specification: 5 - Versioning of Serializable Objects (oracle.com)](https://docs.oracle.com/en/java/javase/17/docs/specs/serialization/version.html#compatible-changes)
* Deserialization işlemi constructor vesaire çağırmaz. Doğrudan readObject() methodu ile değişkene atama yapar. Bu yüzden field tipi ve ismi değiştirmemek önemlidir aksi durumda compatible problemleri ortaya çıkar.
* Kendi readObject() ve writeObject() methodlarımızı yazarak compatible problemlerinin üstesinden gelebiliriz.

**Transient**

* **Serialization** işleminde görmezden gelinecek alanlarda kullanılan keyword’dur.
* Hassas bilgiler içeren alanlarda veya sürekli değişen kalıcı olması gerekmeyen değişkenler için kullanılabilir.

**serialVersionUID**

* Bir runtime fieldıdır, eğer açıkça yaratılmadıysa örtük olarak compiler tarafından serializable classlar için yaratılır. Field sayısı, türleri ve tanımları gibi detayları barındırır.
* Bir streamden nesne okurken, nesnede saklanan serialVersionUID kontrol edilir. Bu kontrol işleminde dosyada (stream) saklanan nesnedeki serialVersionUID ile derlenen sınıf dosyasında bulunan nesne karşılaştırılır. Eğer eşleşmezlerse bir uyumluluk sorunu mevcuttur ve ***InvalidClassException*** fırlatılır.  
  + Ek olarak bu hata farklı compilerlarda da oluşabilir. Farklı compilerlar farklı serialVersionUID’ler üretebilirler. Örneğin eski bir java sürümünde yazılmış bir dosyayı farklı bir sürümde okumaya çalışırken bu hata oluşabilir.  
      
    Bunu önlemek için class’ın içerisine kendi oluşturduğunuz serialVersionUID’i eklemeniz ***şiddetle tavsiye edilir.***



Concurrency/Threads

* Bir process tarafından oluşturulan her threads bu process’in heap’ine erişebilir. Yani thread’ler process’in belleğini ve dosyalarını paylaşırlar. Bu sebeplede dikkatli olunmazsa büyük sorunlar ortaya çıkabilir.
* Her threads’in ***thread stack*** adında kendi heap’leri de bulunmaktadır. Bu ***thread stack***’e yalnızca sahibi olan thread erişebilir.
* Bir kod satırından veya fonksiyondan gelecek bir şeyi beklerken farklı şeylerde yapmak isteyebiliriz. Örneğin bir web sitesinden ve bir veritabanından veri alacağız. Thread kullanmazsak birinden gelecek veriyi bekleyip daha sonra diğerinden veri gelmesini beklememiz gerekecek. Bunun yerine bunu thread’ler ile yapıp iki işlemi de farklı thread’ler ile aynı anda yapabiliriz.
* Java'da thread kullanarak paralel işlemler yürütebilirsiniz. Ancak bu threadlerin ne zaman çalışacağı ve hangi sırayla çalışacakları, JVM ve işletim sistemi tarafından belirlenir. JVM, threadleri oluşturur, başlatır ve sonlandırır, ancak işletim sistemi bu threadlerin fiziksel olarak hangi processte ve hangi işlemci çekirdeğinde çalıştığını kontrol eder. Threadlerin çalışma sırasını veya zamanlamasını tam olarak kontrol etmek zor olabilir ve bazen bu konuda sınırlamalara tabi olursunuz. Bu nedenle, threadlerle çalışırken, threadler arasında senkronizasyon, iş parçacığı öncelikleri ve benzeri konuları yönetmek önemlidir.

**Thread**

* **Thread** classını kalıtım aldığımız classlarda run methodunu override ederek kendi thread’ımızı yazabiliriz.
* Thread’ı başlatmak için override ettiğimiz run methodu yerine, run methodunu çağıran start methodunu kullanırız. Bunun sebepleri;  
  + start metodu, yeni thread oluşturur ve bu thread’i çalıştırmak için işletim sistemine talepte bulunur. Ayrıca, Java'nın thread yönetimini sağlayan altyapıları etkinleştirir.
  + start metodunun kullanılması, farklı threadlerin aynı anda veya eşzamanlı olarak çalışmasını sağlar. run metodu ise yalnızca mevcut thread içinde sıradan bir metot gibi çalıştırılır ve bu, yeni bir iş parçacığı oluşturmaz veya paralel çalışmayı başlatmaz.
* Bir Thread nesnesinin start methodunu yalnızca bir defa çağırabiliriz. Bu sebeple Anonymous class olarak da oluşturmak mantıklı olabilir.

metin, ekran görüntüsü, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu

* Thread’lerin çalışma sırası JVM ve işletim sistemine bağlıdır ve bu sıra değişebilir.  
    
  metin, ekran görüntüsü, yazılım, multimedya yazılımı içeren bir resim

  Açıklama otomatik olarak oluşturuldu
* Thread.setName() methodu ile bir thread’e isim verilebilir.
* **Thread.sleep()** methodu ile bir thread’i uyutabiliriz. Farklı bir thread onu interrupt ederse uyanabilir. Beklenecek süre milisaniye ve nanosaniye cinsinden verilir, eğer işletim sisteminin istediği ayrıntı düzeyini desteklemiyorsa tam istenilen süre kadar beklenmeyebilir. Örneğin nanosaniyeleri desteklemiyor olabilir.

**Runnable ile Thread Oluşturma**

* **Runnable** bir ***FunctionalInterface***’dir. Bizden run() methodunu implemente etmemiz beklenir. Bu interface’i implemente ettiğimiz sınıfın bir nesnesini Thread constructuruna yollayarak bir Thread elde edebiliriz.

**Interrupt**

* Bir thread’in çalışmasını durdurma veya kesme amaçlı kullanılan bir yöntemdir. Bir thread diğer bir thread tarafından kesilmek istendiğinde veya bir thread beklenmedik bir şekilde uzun süre çalışıyorsa, interrupt kullanılarak bu thread’in çalışmasını sonlandırmak mümkün olabilir.
* **Interrupt Flag:** Bir Thread.interrupt() çağrısı yapılması, thread’in interrupt flag’ini ayarlar. Thread bu flag’i kontrol ederek çalışmasını durdurabilir veya devam edebilir.
* **Thread.sleep(),** **Object.wait(),** veya **java.nio.channels.InterruptibleChannel** gibi bazı işlem ve yöntemler, bir thread’e interrupt isteği (normal çalışmasını kesmek, beklemek veya uyumak gibi)gönderir. Eğer thread interrupt isteği sırasında kesintiye uğrarsa (Thread.interrupt()), **InterruptedException** adında bir istisna (exception) fırlatılır.
* Thread **Thread.interrupt()** isteğine yanıt vermezse, bayrak sadece ayarlanmış olur ve thread parçacığı uyumaya veya beklemeye devam edebilir.
* Thread, interrupt flag’ini kontrol ederek, isteğe göre çalışmasını durdurabilir veya kesilme işaretini görmezden gelebilir. Interrupt flag’ine yanıt vermek thread kodunun sorumluluğundadır.
* **Thread.interrupt()** kullanırken dikkat edilmesi gereken önemli bir husus, thread kodunun interrupt flag’ini nasıl ele alacağının iyi düşünülmesi gerektiğidir. Kesme isteği geldiğinde iş parçacığının güvenli bir şekilde sonlanması veya kaynaklarını serbest bırakması önemlidir.

**Join**

* İki adet thread’imiz olsun. Birinci thread ağdan veri alsın ikinci thread ise bu veriler üzerinde değişiklik yapsın. Bu durumda birinci thread bitmeden ikinci thread’in başlamamasını isteriz. Bunu da **Thread.join()** ile sağlarız.
* Peki bir hatadan dolayı birinci thread hiç bitmezse? Bu durumda program uykuda veya donmuş gibi görünecek ve hiçbir şey yapmayacaktır. Bunu da **Thread.join(long millis)** ile engelleyebiliriz. Birinci thread biterse veya millis kadar süre geçerse ikinci thread devam eder.
* Bu işlemi bir komuttan ziyade JVM ve işletim sistemine bir öneri olarak görmek en doğrusudur. Önerimiz uygulanabilir de uygulanmayabilir de. Threadlerin çalışması tamamen JVM ve işletim sistemine bağlıdır.

**Multithreading**

* Aynı anda birden fazla **thread’in** çalışmasıdır.
* Aynı anda birden fazla **thread** çalışırken, bazı durumlarda aynı bellek adresine erişmeye veya değiştirmeye çalışmak isteyebilirler. Bu da hatalı davranışlara yol açabilir. Bu duruma ***Race Conditions*** denir.
* **Thread interference**, multithreading programlamada, birden fazla iş parçacığının aynı anda paylaşılan verilere erişerek ve bu verileri değiştirerek neden olabileceği bir senaryoyu ifade eder. Şu iki temel durumda ortaya çıkar;
  + **Race Condition:** Birden fazla thread, aynı veriye aynı anda erişmeye çalışır ve bu veriyi okuma veya yazma işlemleri gerçekleştirir. Eğer uygun senkronizasyon mekanizmaları kullanılmazsa, bu durum race condition'a yol açar ve verilerin tutarsız veya hatalı bir şekilde güncellenmesine neden olabilir.
  + **Concurrent Modification**: Bir thread, bir veri yapısını (örneğin, liste veya map gibi) değiştirirken, diğer thread’ler de aynı veri yapısına erişip onu değiştirmeye çalışırsa, concurrent modification (eş zamanlı değişiklik) sorunu ortaya çıkabilir. Bu durumda, veri yapısı bozulabilir veya beklenmeyen sonuçlar elde edilebilir.
* Bir sınıf veya methodun ***Thread Safe*** olması onun tüm kritik bölümlerinin senkronize edildiği ve multithread işlem sırasında veri bütünlüğünün ve tutarlılığının koruyacağını belirtir.
* **metin, yazı tipi, ekran görüntüsü, kara tahta içeren bir resim

  Açıklama otomatik olarak oluşturulduSyncronized** anahtar kelimesi ile bir methodun veya kritik bölgenin aynı anda yalnızca bir thread tarafından kullanılması sağlanabilir.  
    
  **metin, yazı tipi, ekran görüntüsü, çizgi içeren bir resim

  Açıklama otomatik olarak oluşturuldu**

**Syncronized**

* Java'da bir metodu veya kod bloğunu "synchronized" olarak işaretlemek, bu metodu veya kod bloğunu çalıştırmak isteyen threadler arasında bir nesne kilidi (object lock) oluşturur.
* Synchronized bir metodun veya kod bloğunun bir sınıfın bir nesnesine uygulanması durumunda, bu nesne üzerinde bir kilit mekanizması oluşturulur. Buna ***Object Locking*** denir.
* Nesne kilitleri adı üstünde nesne bazlı oluşturulur.
* Synchronized alanlar için bir nesne kilidi oluşturulur ve bir thread bu synchronized alanlara erişmeye çalıştığında nesne kilidinin onda olup olmadığı kontrol edilir. Thread nesne kilidine sahipse o alana girebilir yoksa sıraya girer ve kilit ona geçene kadar bekler. Nesne kilidi mekanizması yalnızca synchronized alanlar için geçerlidir, yani bir thread synchronized bir alana erişse ve nesne kilidini alsa aynı anda ikinci bir thread non-synchronized bir alana erişmeye çalışsa ikiside erişim sağlayabilir ve yine hatalar meydana gelebilir.
* Eğer static bir alan synchronized edilirse. Kilit nesne bazlı değil sınıf bazlı olur.
* Yapılabilecek en az kod bloğunu syncronized yapmak en doğrusudur. Aksi takdirde thread’ler gereksiz yere askıya alınır.
* Yalnızca syncronized alanlar içerisinde çağrılabilen fonksiyonlar vardır.  
  Bunlar; ***wait, notify, notify all’****dır.* Amaçları ***Deadlock***durumunu önlemektir.
* *Synchronized* bloğunun dezavantajları da bulunmaktadır.
  + Nesne kilidini bekleyen engellenen thread’ler interrupt edilemez.
  + Synchronized bloğun aynı method içerisinde olması gerekir. A methodu synchronized olsun ve B methodunu çağırsın, nesne kilidi B methodunda serbest bırakılamaz.
  + Bir nesne kilidinin kullanılabilir olup olmadığı kontrol edilemez.
  + Birden fazla thread bir nesne kilidi bekliyorsa, ilk gelen nesne kilidini alır diye bir durum yoktur, yani bir sıraya tabii tutulmazlar. Bu da uzun beklemelere veya sorunlara yol açabilir.  
      
    *Bu sebepler yüzünden javaya 1.5 sürümünde* ***java.util.concorrent*** package’i eklenmiştir. Synchronization işlemi class ile yapılabilmektedir.

**Deadlock**

* Multithreading programlamada karşılaşılan bir senkronizasyon sorunudur. Threadler’in birbirini beklemeleri ve nesne kilitlerini hiçbir zaman elde edememelerine denir. Örneğin A ve B Threadleri olsun ve A thread’i x nesnesinin kilidine sahip olsun ve B thread’i y nesnesinin kilidine sahip olsun. Eğer A thread’i devam etmek için y nesnesinin kilidine ve B thread’i de x nesnesinin kilidine ihtiyaç duyarsa ikisi de hareket edemez.
* Deadlock genelde şu dört koşulun aynı anda gerçekleştiği durumlarda meydana gelir;  
  + Karşılıklı Dışlama (Mutual Exclusion): Bir kaynağa sadece bir iş parçacığının erişebilmesi gerektiği durumda, iş parçacıkları bu kaynağı kilitler ve başka iş parçacıklarının erişimine izin vermezler.
  + Bekleme ve Tutma (Hold and Wait): Bir iş parçacığı bir kaynağı kilitlediğinde ve başka bir kaynağa ihtiyaç duyduğunda, bu iş parçacığı diğer kaynağı beklerken kilitlenmiş kaynağı tutar.
  + Döngü (Circular Wait): İş parçacıkları arasında bir döngü oluşur, yani her bir iş parçacığı diğer bir iş parçacığının sahip olduğu kaynağı bekler. Örneğin, iş parçacığı A kaynağa X kilitlerken, iş parçacığı B kaynağa Y kilitler ve iş parçacığı A Y kaynağına erişmek için beklerse, bir döngü oluşur.
  + Kaynakların Sadece Tek İş Parçacığı Tarafından Serbest Bırakılabilmesi: Kaynaklar sadece sahipleri tarafından serbest bırakılabilir. Yani bir iş parçacığı bir kaynağı kilitlerse, yalnızca o iş parçacığı bu kaynağı serbest bırakabilir.
* Deadlock’lar thread’lerin nesne kilitlerini elde etmeye çalıştıkları sıra meydana gelmektedir. Kod senkronize edilirken, bu durumun ortaya çıkabileceği yerlere karşı tetikte olmak gerekir.
* Deadlock’un meydana gelmesini önlemek için nesne kilitleri aynı sırada elde edilmeye çalışılabilir. Örneğin thread1 lock2 ve lock1 şeklinde sırayla kilitleri istiyorsa diğer yazılan threadler de lock2, lock1 sırasıyla kilitleri isterse deadlock oluşamaz.
* **wait():** Genellikte bir koşula bağlı olarak kullanılır ve nesne kilidi tutan thread’in kilidi bırakmasını ve uyandırılana kadar beklemesi belirtir.
  + Thread notify() dışında örneğin işletim sistemi tarafından farklı bir sebepten de uyanabileceği için while ile kontrol sağlamak gerekir. Eğer uyumaya devam etmesi gerekiyorsa while döngüsünden çıkamayacağı için tekrar döngünün başına döner ve tekrar wait() fonksiyonu çağırılır.  
      
    metin, yazı tipi, ekran görüntüsü içeren bir resim

    Açıklama otomatik olarak oluşturuldu
* **notify():** bir nesne üzerinde *wait()*methodu ile bekleyen threadlerden bir tanesini uyandırmak için kullanılır.
  + Hangi thread’in uyandırılacağı tamamen JVM ve işletim sistemi arasındadır. Tahmin edilemezdir.
  + *notify()* kullanılan thread nesne kilidini **serbest** **bırakmaz**. Kilidi elinde tutmaya devam eder fakat wait() ile bekleyen bir thread’i uyandırır. Uyanan thread nesne kilini almak için *notify()* kullanılan thread’in nesne kilidini serbest bırakmasını bekler.
* **notifyAll():** *notify()* methodu ile tamamen aynıdır. Tek fark *notify()* methodu rastgele bir thread’i uyandırırken *notifyAll()* methodu wait ile bekleyen tüm thread’leri uyandırır. Bu thread’ler nesne kilidi için yarışırlar. *notifyAll()* methodunun çağırıldığı thread nesne kilidini bırakınca diğer thread’ler almaya başlar.
  + Çok sayıda thread ile çalışmıyorsak notifyAll() kullanmak geleneksel kullanımdır. Fakat çok sayıda thread ile çalışıyorsak bu büyük bir performans sorununa yol açabilir.

**Atomik İşlemler**

* *Atomik işlemler* yapılırken thread işletim sistemi tarafından askıya alınamaz.
* Atomic olarak kabul edilen işlemler;
  + Referans değişkenleri okuma (object1 = object2), primitive türler okunurken veya yazılırken (long ve double hariç çünkü onlar 64 bittir ve 64 bit oldukları için onlar üzerinde fazladan işlem yapılabilir. Örneğin 32 bit mimaride kullanılabilmek için iki ayrı işlem gerekir.)
  + **volatile** anahtar kelimesi ile işaretlenmiş değişkenlerin okuma ve yazma işlemleri de atomik olarak kabul edilir.
    - Örneğin counter++ diye bir işlemimiz olsun. Bu işlem için adım adım işlemler şu şekildedir.

1-) counter değerini ana bellekten oku.

2-) değere 1 ekle.

3-) değeri ana belleğe yaz.

Eğer değişken volatile değilse ana belleğe verinin ne zaman geri yazılacağı JVM’e bağlıdır. Farklı bir thread’den erişmeye çalışıldığında sorun yaşanabilir (önceki değeri okuma gibi). Bunu ***volatile*** anahtar kelimesi ile önlüyoruz. ***volatile*** olduğunda tüm işlemler aynı süre içerisinde başlar yapılır ve bitirilir. Askıya alınma durumu olmaz.

* + - Aynı örnek için değişken volatile olsa ve iki thread aynı anda onu artırmaya çalışsa yine sorunlar ortaya çıkabilir. Örneğin thread1 bellekten değeri okur ve 1 olsun, aynı anda thread2 de bellekten değeri okur yine 1, sonra thread1 değeri artırır, thread2 de değeri artırır. Bu durumda ikisinin elinde de 2 değeri bulunuyor ve bunu ana belleğe yazarlar. İşlem sonucunda değerimiz 2 olur fakat 3 olması gerekiyordu. Burada da yine synchronization mekanizmaları kullanılması gerekir.
  + Bazı atomic sınıflar, java.util.concurrent.atomic gibi sınıflar Atomic başlığıyla başlayan sınıflar sunar.
  + ***Atomic sınıflar***, çoklu iş multithreading programlamada paylaşılan değişkenlerin atomik işlemlerle güvenli bir şekilde değiştirilmesini ve güncellenmesini sağlamak için kullanılan sınıflardır. Bu sınıflar, senkronizasyon mekanizmalarını kullanmadan paylaşılan değişkenler üzerinde işlem yapma yeteneği sağlarlar ve böylece thread-safe kod yazmayı kolaylaştırırlar.

**java.util.concurrent package**

**ReentrantLock Class**

* Bu sınıfla synchronized bloğun yaptıklarını daha esnek bir şekilde yapabiliriz.
* **ReentrantLock.lock()** methoduylanesnenin kilidini alabiliriz. Eğer kilit alınamaz durumdaysa thread kilidi almak için sıraya girer ve askıya alınır. Bir nesne kilidi lock() ile defalarca alınabilir, örneğin bir method içerisinde iki defa lock() methodunu çağıralım ve bir kere unlock() methodunu çağıralım. Method bittiğinde nesne kilidi bırakılmaz. Özellikle döngü içerisinde lock() kullanılan noktalarda buna dikkat edilmelidir.
* **ReentrantLock.unlock()** methoduyla nesnenin kilidini serbest bırakabiliriz.

*Kilit alma ve bırakma işlemleri bu sınıfta tamamen yazılımcının elindedir, otomatik olarak hiçbir şey yapılmaz****.***

* Kapama işlemlerini try-catch-finally ile yapmak iyi bir yoldur.
  + Okunabilirlik artar ve lock-unlock kısmı daha derli toplu olur.
  + Kritik alandan bir exception fırlatılırsa ve bu exception’u catch ile yakalayamıyorsak kilitlerin serbest bırakılmasını gerekir.
  + ReentrantLock AutoClosable olmadığından try-witch-resource kullanamayız.
* **ReentrantLock.tryLock()** methodu ile bir nesne anahtarı almayı deneyebiliriz. Eğer alınabilirse true, alınamazsa false döndürür. Bir if bloğuna koyarak alındığında veya alınamadığında işlemler yapılabilir. Bir long ve TimeUnit’i parametre olarak alan overloaded versiyonu da vardır. Kilit alınmaya çalışılırken belirli bir süre almayı bekler, eğer alamazsa false döndürür.   
    
  ****
* Eğer **ReentrantLock(true)** şeklinde constructora true değeri verilirse. Bu adil bir kilitleme sağlanmaya çalışılmasını belirtir. Yani nesne kilidini bekleyen thread’lerden en uzun bekleyenin nesne kilidini almasını sağlar.,

**Thread Pool**

* Multithreading programlamada kullanılan bir design pattern’dir. Thread yönetimini kolaylaştırmak amacıyla kullanılır.
* Belirli bir görev yürüten thread’leri önceden oluşturur ve bunları havuzda saklar.
* Thread’ler önceden oluşturulduğu ve havuzda tutulduğu için işlet başlatma ve sonlandırma maliyeti azalır. Her işlem için thread oluşturmanın getirdiği ağırlık azalır ve uygulamanın daha verimli çalışması sağlanır.
* *Thread Pool,* işleri otomatik olarak thread’lere dağıtır ve thread’ler arasında iş yükünü dengeler. Bu, sistem kaynaklarının daha iyi kullanılmasını sağlar ve işlemlerin daha hızlı tamamlanmasına yardımcı olur.
* Aynı anda çalışacak thread sayısı sınırlanabilir. Bu, thread yaratma işlemi sırasında sistemin veya kanağın kapasitesinin aşılmasını engeller. Aynı zamanda etkin olan, engellenen thread’lerin de sayısı sınırlandırılabilir.
* ***ExecutorService*** sınıfı altında ***newFixedThreadPool(), newCachedThreadPool(), newSingleThreadExecutor()*** gibi factory methodlarla *Thread Pool*’lar oluşturulabilir.
* Thread Pool ile thread sayısı sınırlandırıldığından bir görevi çalıştırmak istediğimizde hemen çalışmayabilir. Örneğin Thread Pool’u maksimum 5 adetle sınırlandırmış olalım ve bir görev gönderelim. Görevi gönderdiğimizde bu 5 thread de aktif ise, gönderdiğimiz görev bu 5 thread’den biri gerçekten sonlanana kadar hizmet kuyruğunda bekleyecektir.

**ExecutorService**

* Multithreading’i kolaylaştırmak için sunulmuş bir Interface’dir.
* Thread üzerinde işlem yapmayı yönetmek ve ***Thread Pool*** oluşturmak gibi görevleri daha kolay bir şekilde gerçekleştirmeye olanak tanır. Thread’leri ve onların yaşam döngülerini yönetmekle uğraşmadan çalıştırmak istediğimiz koda odaklanmamızı sağlar.
* Thread oluşturmanın maliyetini de optimize eder.
* Kullanıldıktan sonra ***Thread Pool***’un **ExecutorService.shutdown()** ile kapatılması gerekir.
* **ExecuterService.shutdown()** methodu tüm thread’lerin kapanmasını bekler ardından pool’u kapatır. Eğer ani bir şekilde her şeyin kapanmasını istiyorsak **ExecuterService.shutdownNow()** methodu kullanılır.
* **ExecuterService.shutdownNow()** methodu pooldaki tüm görevleri durdurmaya çalışır, kuyruktaki tüm görevleri de atar. Ancak bunları yapabileceğinin bir garantisi yoktur. Bu yüzden shutdown() methodunu kullanmak iyi olandır.
* **ExecuterService.execute()** veya **ExecuterService.submit()** methodları ile pool’a görevler gönderilebilir.
* **ExecuterService.execute()**
  + Herhangi bir şey döndürmez, Runnable interface’sini uygulayan bir işlem alır ve onu çalıştırır.
  + Eğer işlem sırasında bir exception oluşursa doğrudan yayınlanır ve havuzdaki thread sonlandırılır.
  + Genellikle işlem sonucunu almanız gerekmediğinde veya işlem sonucuyla ilgilenilmediğinde kullanılır.
* **ExecuterService.submit()**
  + Callable interface’sini uygulayan bir işlem alır ve onu çalıştırıp **Future** nesnesi olarak gönderir. **Future** nesnesi işlemin sonucunu almaya ve hatta işlem tamamlanana kadar beklemeye olanak tanır.
  + Eğer işlem sırasında bir exception oluşursa bu exception **Future** nesnesi içerisinde saklanır ve işlem beklerken get() yöntemi ile exceptionu ele almaya olanak tanır. Bu sayede istisalar thread’i tamamen sonlandırmaz.
  + İşlem sonucu ile çalışılacaksa kullanışlıdır.
  + Runnable interface’sini uygulayan bir işlem alan overloaded versiyonu da vardır fakat kullanılması pek tavsiye edilmez, onun yerine execute() kullanmak daha doğrudur.

**ArrayBlockingQueue**

* Kapasitesi geliştirici tarafından constructorda verilir.
* Thread Safe’dir.
* Sıradan thread alınırken (take() methodu) veya sıraya thread koyulurken (put() methodu) thread’in beklemesini sağlar.   
  + **put()** eğer sıra doluysa, bu metot çağıran thread’i beklemeye alır ve sıra boşaldğında öğeyi ekler.
  + **take()** eğer sıra boşsa, bu metot çağıran thread’i beklemeye alır ve queue’ye öğe eklenene kadar bekler.
  + **offer(),** queue’ye öğe eklemeye çalışır ekleyemezse false döner.
  + **poll(),** queue’den öğe almaya çalışır, ancak queue boşsa null döner.

**metin, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**Thread Starvation**

* multithreading programlama bir thread’in beklemeye sürekli devam ettiği veya kaynaklara erişemediği durumdur. Bu durum, genellikle synchronization, race condition veya kaynak paylaşımı sorunlarından kaynaklanır.
* Şu senaryolarda Thread Starvation görülebilir;
  + **Öncelik Sorunları:** Thread’lerin öncelikleri yanlış ayarlandığında veya yanlış sıralandığında bir thread diğerlerine göre sürekli olarak öncelik kazanabilir ve diğer iş parçacıkları kaynaklara erişim beklemeye devam edebilir.
  + **Yavaş İş Parçacıkları:** Bazı iş parçacıkları, diğerlerine göre daha yavaş çalışıyorsa, diğer iş parçacıkları bu yavaş iş parçacıklarının bitmesini beklemek zorunda kalabilirler.
* **Fair lock** ile *Thread Starvation* engellenebilir.



* Eğer uygulama iki yada üç thread kullanıyorsa, yada thread üzerinde çalıştırılan görevler çok hızlı şekilde rekabet ediyorsa ya da birden fazla thread’in bloke olmasının çok nadir olacağının farkındaysanız, synchronized blok kullanmak daha doğru bir karardır. Çünkü **fair lock** kullanmak performansı etkileyecektir.
* Thread’lerin sürekli olarak birbirine cevap verdiği ancak işlerini tamamlamadıkları bir senaryoya Live Lock denir, örneğin karşı karşıya birbirlerine gelen iki kişinin yol vermek için sürekli aynı adımları atarak ikisinin de ilerleyememesi ve sürekli bir sağa bir sola gitmeleri gibi, çalışır vaziyette olsalar da işlerini ilerletemezler.

metin, ekran görüntüsü, yazılım içeren bir resim

Açıklama otomatik olarak oluşturuldu

JUnit

* Her test methodu birbirinden bağımsız olmalıdır.
* **assertEquals()** ile değer kontrolu yapılabilir.

metin, yazılım, multimedya yazılımı, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

* Test class da değişken tanımlama. Tanımlamalar test methodlarında bulunsun. Fakat sürekli kod tekrarı yapılıyorsa bunu ***@BeforeEach*** annotation ile method yazarak önleyebiliriz.
  + ekran görüntüsü, metin içeren bir resim

    Açıklama otomatik olarak oluşturuldu***@BeforeEach*** annotation’u ile yazılmış methodlar her test methodu çağrısından önce çağrılır. Veritabanı bağlantısı veya belirli nesneleri başlatmak için kullanılabilir.
* ***@BeforeEach*** annotation’u her method çağrısından önce çalışır dedik, peki bir defa çalışmasını istediğimiz bir kod istiyorsak ne yaparız? Bunun için de ***@BeforeClass*** annotation’u kullanılır. Yalnızca bir kez çağrılacak static bir method oluşturulur. Burada veritabanı bağlantısı başlatma gibi işlemler yapılabilir.
* ***@BeforeAll***’ın tam tersi işlemi yapan yani tüm kontroller bittikten sonra bir defa çalıştırılacak bir fonksiyon yazmak istiyorsak da ***@AfterClass*** annotation’u kullanılır. Açılan bağlantıların, dosyaların kapatılma işlemleri yapılabilir.
* ***@AfterAll, @AfterEach*** annotationları @*BeforeEach* ve @*BeforeAll* annotationlarının tam tersidir.
* ***assertThrows()*** methodu ile bir exception da kontrol edilebilir.  
    
  metin, ekran görüntüsü, yazı tipi içeren bir resim

  Açıklama otomatik olarak oluşturuldu
* Farklı değerler ile test yapmak istediğimizde **Parametized Test** yazabiliriz.   
  metin, ekran görüntüsü, yazı tipi, yazılım içeren bir resim

  Açıklama otomatik olarak oluşturuldu

Database

**SQLite**

* Sqlite’da sütunlar için tip belirtsek de aslında herhangi bir sütuna her türlü veri konulabilir.
* .backup ile yedek alınabilir.
* Primary key Id alanlarını “\_id” şeklinde tanımlamak iyi bir davranıştır. Bazı java sınıfları “\_id” adında bir id sütunu gerektirir.
* ResultSet.get methodlarında columnName yerine index kullanmak daha hızlıdır.

**View**

* Bir veya daha fazla tabloyu temel alarak oluşturulan ve bir tabloyu taklit eden, ancak fiziksel bir veri deposuna sahip olmayan bir sanal tablodur. View, tabloyu sorgulamak veya belirli bir şekilde şekillendirmek için kullanılır ve daha karmaşık sorguları basitleştirmek veya belirli bir bakış açısını sunmak için kullanışlıdır.
* Sanal Tablo: View, fiziksel bir tabloyu temsil etmez. Yani, gerçek veri saklamaz. Sadece bir sorgunun sonucunu temsil eden bir "sanal" tablodur.
* Tablo Bağımsızlığı: View, verilerin nasıl saklandığını veya nasıl düzenlendiğini gizler ve bu nedenle uygulamaların sadece view'ı sorgulamasına izin verir. Bu, veri modeli değiştiğinde uygulamalarda değişiklik yapma ihtiyacını azaltabilir.
* Veri Güvenliği: View'lar, belirli kullanıcıların yalnızca belirli sütunları veya satırları görmelerine izin vermek için kullanılabilir. Veri güvenliği açısından oldukça önemlidir.
* Sorgu Basitleştirmesi: Birden çok tabloyu içeren karmaşık sorguları basit hale getirmek için view'lar kullanılabilir. Karmaşık sorguları view'lar içinde tanımlayarak, uygulama kodunu temiz ve anlaşılır tutabilirsiniz.
* Veri Düzenlemesi (Updateable View): Bazı veritabanı sistemleri, belli koşulları sağlayan view'ları düzenlenebilir yapar. Bu sayede view üzerinden veri eklemek, güncellemek veya silmek mümkün olabilir. ***Oracle Database***, ***Microsoft SQL Server*** updateable view’lara izin verirken **SQLite** izin vermez.

**SQL Injection**

* Kullanıcı girişi alınıp datasabeden veri vs çekilen bir yerden (console, textbox, vs) sql sorgusu yazılarak databaseye ulaşılıp, databaseyi manipüle ederek yapılan saldırıdır.
* SQLite JDBC sürücüsünde **SQL Injection** otomatik olarak engellenir çünkü Statement.execute() ve Statement.executequery() methodları yalnızca bir SQL ifadesini çalıştırırlar. Örneğin “;DROP TABLE tablename” şeklinde ikinci bir ifadeyi çalıştırmazlar.
* Önlemenin bir yolu da javada bulunan ***PreparedStatement*** sınıfını kullanmaktır.

**PreparedStatement**

* veritabanı işlemleri için kullanılan SQL sorgularını önceden derler ve bu sorguları veritabanına gönderirken yerine değerleri yerleştirmenizi sağlar.
* JDBC’nin bir parçasıdır, ilişkisel veritabanlarına erişmek için kullanılır.
* Temel avantajları;  
  + **Ön Derleme (Precompilation):** PreparedStatement nesneleri, SQL sorgularını veritabanına göndermeden önce veritabanı tarafından ön derlenir. Bu, sorgunun daha hızlı çalışmasına yardımcı olabilir, çünkü her sorgu her seferinde yeniden derlenmez.
  + **Güvenlik:** PreparedStatement, SQL Injection saldırılarına karşı daha dirençlidir çünkü sorgu parametreleri yerine geçirilir ve otomatik olarak temizlenir. Bu, kullanıcı tarafından girilen verilerin güvenli bir şekilde işlenmesine yardımcı olur.
  + **Performans:** PreparedStatement nesneleri, aynı sorguyu farklı değerlerle birden çok kez çalıştırırken performans avantajı sağlar. Parametre değerlerini sadece değiştirerek aynı sorguyu tekrar tekrar kullanabilirsiniz.
  + **Okunabilirlik:** PreparedStatement, parametreleri sorguya yerleştirirken daha okunaklı ve sade bir şekilde kod yazmanıza olanak tanır.

**metin, ekran görüntüsü, yazılım, multimedya yazılımı içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**Transactions**

* Veritabanında bir veya daha fazla SQL sorgusunu bir araya getirerek bir işlem olarak gruplandırma ve bu işlemi ya tamamen gerçekleştirme ya da hiç gerçekleştirmeme yeteneğini ifade eder.
* Transaction’lar ACID uyumlu olmalıdır.
  + ***A*tomicity (Atomiklik):** Bir işlem içindeki tüm SQL sorguları bir arada çalışır ve ya hepsi başarılı bir şekilde tamamlanır ya da hiçbiri çalışmaz. Herhangi bir aşamada hata olursa, tüm işlem geri alınır (rollback) ve veritabanı başlangıç durumuna geri döner.
  + ***C*onsistency (Tutarlılık):** İşlem başarılı bir şekilde tamamlandığında, veritabanı tutarlı bir durumda olmalıdır. Yani, veritabanı kendi içsel kural ve bütünlük kriterlerini korur.
  + ***I*solation (İzolasyon):** Birden fazla işlem aynı anda çalıştığında, bir işlem diğerlerinden bağımsızdır. Yani, bir işlem diğer işlemlerin işlem sırası ve sonuçlarından etkilenmez.
  + ***D*urability (Süreklilik):** İşlem tamamlandığında, veritabanı değişiklikleri kalıcıdır ve sistem kapansa bile veriler kaybolmaz.

Networking Programming

* Javadai istemci soketi için **Socket** ve sunucu soketi için **ServerSocket** classları bulunmaktadır.
* 0 – 65535 arası bir sayı port olarak seçilebilir. Eğer ***ServerSocket serverSocket = new ServerSocket(port);*** kodundan bir hata dönmezse port başarılı bir şekilde açılmış anlamına gelir. Eğer hata dönerse portun kullanılamaz olduğu anlaşılır.

**URI**

* URI, bir kaynağın benzersiz bir tanımlayıcısıdır. URI, kaynağın adını veya konumunu ifade etmek için kullanılır, ancak bu konumun fiziksel olarak erişilebilir olması gerekmez. URI, URL ve URN (Uniform Resource Name) olmak üzere iki temel alt kategoriye ayrılır.

**URL**

* URL, bir kaynağın konumunu belirtir ve bu konumu doğrudan erişilebilir bir şekilde ifade eder. URL, bir kaynağın nerede bulunabileceğini belirten tam bir adresi içerir. Örneğin, "https://www.example.com" bir URL'dir ve web tarayıcıları gibi uygulamalar bu adresi kullanarak kaynağa doğrudan erişebilir.

***URI, tanımladığı kaynağa erişmek için yeterli bilgi sağlamayabilecek bir tanımlayıcıdır. Öte yandan bir URL, tanımladığı kaynağa nasıl erişileceği hakkında bilgi içeren bir tanımlayıcıdır.***

***Javada tavsiye edilen kullanım önce URI olarak kullanıp ardından gerektiğinde URL’e dönüştürmektir.***

**URLConnection**

* **URL.openStream()** ile bir URL’den veri okunabilir. Fakat bu işlemi daha fazla özellik sağlayan **URLConnection** ile yapmak daha doğrudur.

Modüller

* Bir veya daha fazla package’ı bir araya getiren ve bu package’leri başka modüllere açık veya kapalı bir şekilde ifade eden yapıya denir.
* Java modülleri, daha büyük ve karmaşık uygulamaları modüler hale getirerek kod organizasyonunu ve bakımını kolaylaştırır. Ayrıca, modül sistemleri, istemci uygulamalarının gereksinim duyduğu sınıfları daha iyi kontrol etmeye ve gereksiz sınıfların kullanımını engellemeye yardımcı olabilir.
* Her modül kendi modül tanımlayıcı dosyası ile gelir.
* Modüller **normal modules** ve **open modules** olarak ikiye ayrılırlar.
  + **Normal Modules:** Derleme zamanında ve çalışma zamanında yalnızca açıkça dışa aktarılan paketlerdeki türlere erişim sağlar.
  + **Open Modules:** Derleme zamanında yalnızca açıkça dışa aktarılan paketlerdeki türlere erişim sağlar. Ek olarak tüm paketler dışa aktarılmış gibi tüm paketlerindeki türlere çalışma zamanında erişim izni de verir.
* Her modül belirli modül anahtar sözcüğü kullanılarak tanımlanır.
* Modülleri package’larla aynı adlandırmak iyi bir uygulamadır.

**module-ingo.java**

* **requires** anahtar kelimesi bir modülün başka bir modüle bağımlılığını ifade eder. Bir modül, diğer bir veya birden fazla modülün sağladığı sınıf veya arayüzleri kullanmak istediğinde kullanılır.
* **exports** anahtar kelimesi, bir modülün içindeki paketlerin başka modüller tarafından kullanılabilir olmasını sağlar. Başka bir deyişle, bir paketi başka modüllerin kullanmasına izin vermek için exports kullanılır.
  + **exports … to** ifadesi ile paketi yalnızca belirli bir modüle veya modüller grubuna açmak için kullanılır.

**exports** packageName **to** otherModule1, otherModule2, …;

* **opens** anahtar kelimesi ile **Reflection** ile erişilmesi belirlenir, yani belirlenen modül dışındaki modüller **Reflection** ile erişemez.

**opens** packageName **to** otherModule1, otherModule2, …;

* **transitive** anahtar kelimesi ile bir modülün diğer modüllere olan bağımlılığı aktarılabilir.

A modülü B modülünü ve C modülünü kullanıyorsa, B modülü de C modülünü kullanıyorsa. A modülü C modülüne B modülüne erişerek gidebilir.