



# **Bölüm 5: Koleksiyonlar**

## **JAVA ile Nesne Yönelimli Programlama**



# Java Koleksiyonları

- Veri yönetimi ve işleme konusunda esnek ve etkili çözümler sunar.
- Birden çok öğeyi depolayan ve yöneten veri yapılarıdır.
- Veri toplamak, saklamak, sıralamak ve işlemek için güçlü araçlar sağlar.
- Performans ve güvenlik açısından optimize edilmiş çeşitli koleksiyon türleri sunar.
- Veri yapısını seçerken ihtiyaca uygun koleksiyon türünü seçmek önemlidir.



# Koleksiyon Türleri

- **List:** Sıralı, indeksli öğeleri içerir. (ArrayList, LinkedList)
- **Set:** Benzersiz öğeleri içerir. (HashSet, TreeSet)
- **Map:** Anahtar-değer çiftlerini içerir. (HashMap, TreeMap)
- **Queue:** İlk giren, ilk çıkan (FIFO) veri yapısını sağlar. (LinkedList, PriorityQueue)

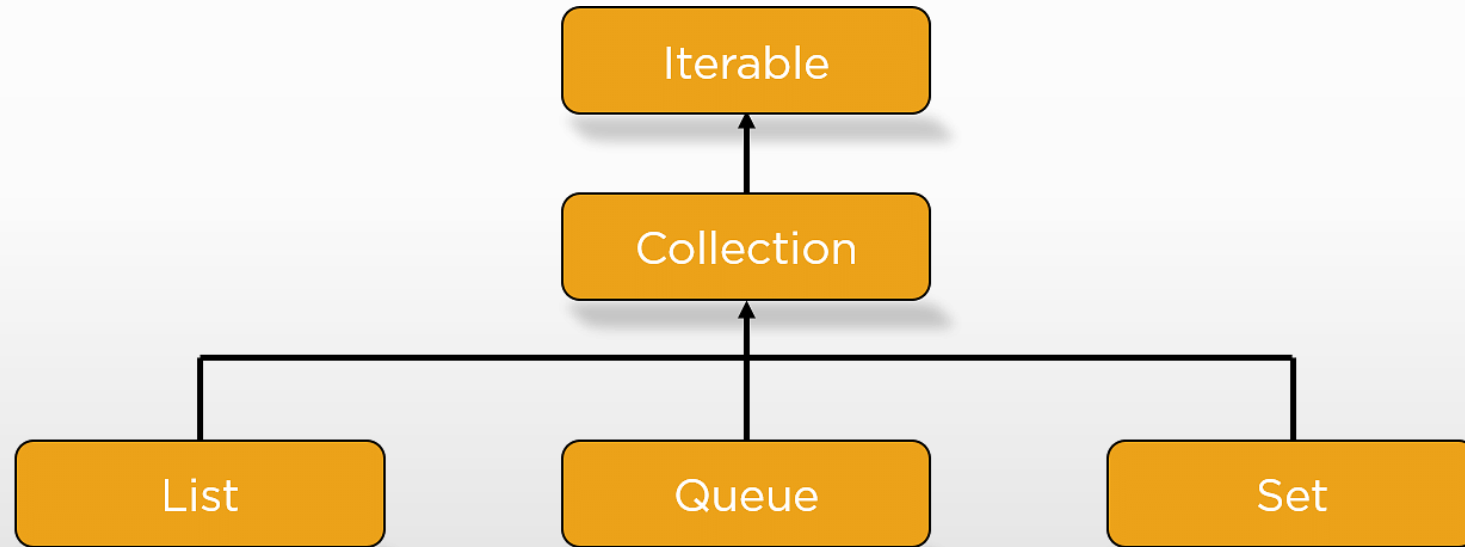


# Koleksiyon İşlemleri

- Ekleme, silme, güncelleme, arama gibi temel işlemleri sağlar.
- forEach döngüsüyle kolay iterasyon imkanı sağlar.
- Koleksiyon türleri, veri hacmine ve erişim ihtiyaçlarına göre optimize edilmiştir.
- Bellek yönetimi ve performans açısından etkili kullanım sağlar.

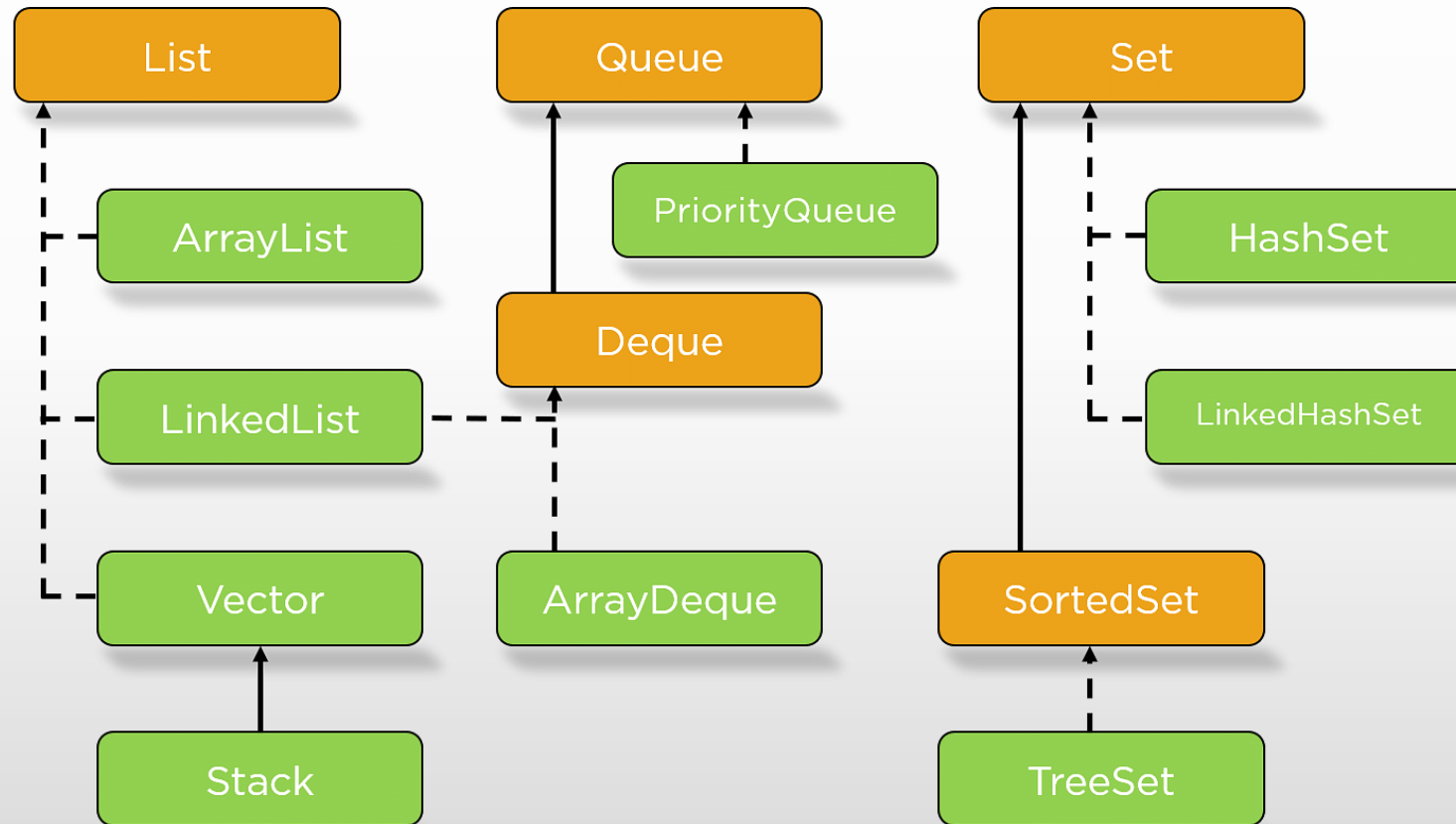


# Java Koleksiyonları Arayüzleri





# Java Koleksiyonları Sınıfları





# Java Koleksiyonlarının Tasarım Amaçları

- Kullanımı basit ve anlaşılır bir API aracılığıyla **kolaydır**.
- Yüksek performanslı veri depolama ve erişim sağlamak üzere **optimize** edilmiştir.
- Generic yapılar kullanılarak, koleksiyonlarda **tip güvenliği** sağlanır.
- Çeşitli veri yapılarına ve ihtiyaçlara uygun **geniş bir sınıf hiyerarşisine** sahiptir.
- Java'nın standart kütüphanesine **entegre** edilmiş API'ler ile **uyumludur**.
- **Güvenli** veri erişimi ve manipülasyonu sağlamak üzere tasarlanmıştır.



# Java Koleksiyonlarının Tasarım Amaçları

- Çeşitli koleksiyon türleri ve alt sınıflar, farklı veri yapılarına **uyum** sağlar.
- Temiz ve düzenli bir API, kodun **anlaşılabilirliğini** artırır ve bakımını kolaylaştırır.
- Koleksiyonlar üzerinde birçok **hazır algoritma** bulunur, bu da geliştiricilere güçlü araçlar sağlar.
- **Hata durumlarını** ele alacak şekilde tasarlanmıştır, bu da daha güvenilir yazılımların oluşturulmasına katkıda bulunur.
- Veri yapıları üzerinde yaygın işlemleri gerçekleştirmek için kullanılan fonksiyonlar, geliştiricilere **zaman kazandırır**.





# Koleksiyon Çerçevesi Unsurları

- **Arayüzler**, kodun genel yapısını belirler ve birlikte çalışabilirliği sağlar.
- **Uygulamalar**, belirli bir veri yapısının detaylarını uygular.
- **Algoritmalar**, çeşitli işlemleri yürütmek için genel stratejiler sağlar.
- **Modüler tasarım**: Her bileşenin belirli bir rolü ve sorumluluğu vardır.
- **Esneklik**: Farklı uygulamalar ve algoritmalar, aynı arayüzle birlikte kullanılabilir.



# Koleksiyon Çerçevesi Unsurları

- **Arayüz (Interface):**
  - Belirli bir veri yapısının nasıl davranması gerektiğini tanımlar.
  - **Örnek:** List arayüzü, bir sıralı listeyi temsil eder.
- **Uygulama (Implementation):**
  - Arayüzleri uygulayan gerçek sınıflardır.
  - Veri yapılarının özel uygulamalarını içerirler.
  - **Örnek:** ArrayList sınıfı, List arayüzünü uygular, dinamik bir dizi sağlar.
- **Algoritma (Algorithm):**
  - Sıralama, arama gibi işlemleri gerçekleştiren algoritmaları içerir.
  - **Örnek:** Collections.sort() fonksiyonu, bir koleksiyonu sıralamak için kullanılır.



# Arayüz ve Uygulama İlişkisi

- Arayüz, belirli bir davranışı tanımlar. Sınıflar bu davranışı uygular.
- Farklı Depolama Türleri:
  - ArrayList: Dinamik bir diziyi temsil eder.
  - HashMap: Anahtar-değer çiftlerini depolar.
  - LinkedList: Çift yönlü bağlı bir liste olarak kullanılabilir.
- Her sınıf, belirli bir veri yapısının gereksinimlerini karşılamak üzere tasarlanmıştır.
- **Modüler tasarım:** Her sınıf, belirli bir görevi yerine getirir.
- **İstendiğinde değiştirilebilirlik:** Bir arayüz değiştirilebilir, ancak temel davranış korunur.



# Kullanım Örnekleri

- İsim Listesi (ArrayList):

```
List<String> isimListesi = new ArrayList<>();  
isimListesi.add("Ahmet");  
isimListesi.add("Ayşe");
```

- Telefon Rehberi (HashMap):

```
Map<String, String> telefonRehberi = new HashMap<>();  
telefonRehberi.put("Ahmet", "555-1234");
```



# Arayüz Metotları

- Java Koleksiyon Arayüzü, koleksiyon sınıflarının ortak metotlarını belirleyen bir arayüzdür.
- Temel Metotlar:
  - **add(E e)**: Belirtilen öğeyi koleksiyona ekler.
  - **remove(Object o)**: Belirtilen öğeyi koleksiyondan kaldırır.
  - **size()**: Koleksiyonun eleman sayısını döndürür.
  - **isEmpty()**: Koleksiyonun boş olup olmadığını kontrol eder.
- Döngü Metotları:
  - **iterator()**: Koleksiyon üzerinde bir iterator döndürür.
  - **forEach(Consumer<? super E> action)**: Her eleman üzerinde belirli bir işlemi uygular.



# Arayüz Metotları

- Diziye Dönüşüm:
  - **toArray()**: Koleksiyonu bir diziye dönüştürür.
- Arama ve Kontrol:
  - **contains(Object o)**: Belirtilen öğenin koleksiyonda olup olmadığını kontrol eder.
  - **containsAll(Collection<?> c)**: Belirtilen koleksiyonun tamamının koleksiyonda olup olmadığını kontrol eder.
- Koleksiyon Temizleme:
  - **clear()**: Koleksiyondaki tüm öğeleri kaldırır.



# List Arayüzü

- Java'da sıralı koleksiyonları temsil eder, indeks tabanlı erişim sağlar.
- Elemanlar sıralı bir şekilde depolanır, her bir elemanın indeksi vardır.
- List Arayüzü uygulamaları:
  - ArrayList: Dinamik bir dizi olarak çalışır, hızlı indeks tabanlı erişim sağlar.
  - LinkedList: Çift yönlü bağlı liste olarak çalışır, elemanlar arasında hızlı eklemeler ve kaldırmalar sağlar.
  - Vector: Thread-safe bir versiyondur, ancak genellikle ArrayList tercih edilir.
- Eleman ekleme ve çıkarma işlemleri sıkça gerçekleşiyorsa LinkedList tercih edilebilir.



# List Arayüzü

- Temel Metotlar:
  - `add(E e)`: Belirtilen öğeyi listenin sonuna ekler.
  - `remove(int index)`: Belirtilen indeksteki öğeyi listeden kaldırır.
  - `get(int index)`: Belirtilen indeksteki öğeyi döndürür.
  - `size()`: Listenin eleman sayısını döndürür.





# ArrayList Sınıfı

- Java'da dinamik bir dizi yapısı sağlayan ve elemanları sırasıyla depolayan temel bir liste sınıfıdır.
- AbstractList sınıfından türetilmiştir ve List arayüzünü uygular.
- Dinamik Boyut: Otomatik olarak boyutunu ayarlar.
- Sırasal Depolama: Elemanlar eklenme sırasına göre tutulur.
- Eleman ekleme ve çıkarma işlemleri hızlıdır.

```
ArrayList<String> meyveler = new ArrayList<>();  
meyveler.add("Elma");  
meyveler.add("Armut");
```



# LinkedList Sınıfı

- Java'da çift yönlü bağlantılı bir liste yapısı sağlayan ve elemanları düğümler aracılığıyla depolayan temel bir liste sınıfıdır.
- AbstractSequentialList sınıfından türetilmiştir ve List arayüzünü uygular.
- Çift Yönlü Bağlantı: Düğüm bir önceki ve bir sonraki düğümlerle bağlantılıdır.
- Esnek Boyut: Eleman ekleme ve çıkarma işlemleri için uygun yapı sunar.

```
LinkedList<String> meyveler = new LinkedList<>();  
meyveler.add("Elma");  
meyveler.add("Armut");
```



# Vector Sınıfı

- Dinamik olarak büyüyen ve çok sayıda iş parçacığı tarafından güvenle kullanılabilen bir veri yapısıdır.
- Vector, List arabirimini uygular ve genişletilebilir bir dinamik dizi sağlar.
- İlk oluşturulduğunda belirtilen kapasiteye sahiptir, ancak elemanlar ekledikçe otomatik olarak genişler.
- Dinamik boyutlandırma özelliği, esneklik ve genişletilebilirlik sağlar.
- Çok sayıda iş parçacığıyla güvenli bir şekilde kullanılabilir.
- Modern uygulamalarda genellikle daha hafif ve performanslı alternatifler tercih edilir (örneğin, ArrayList).



# Vector Sınıfı

- Temel Metotlar:
  - `addElement(E obj)`: Vector'e bir eleman ekler.
  - `elementAt(int index)`: Belirtilen indeksteki elemanı döndürür.
  - `removeElement(Object obj)`: Belirtilen elemanı Vector'den kaldırır.
  - `size()`: Vector'ün boyutunu (eleman sayısını) döndürür.
  - `capacity()`: Vector'ün mevcut kapasitesini döndürür.



# Stack Sınıfı

- LIFO (Last In, First Out) mantığına göre çalışır.
- Elemanların en son eklenen en üstte (top) ve en eski eklenen en altta (base) olacak şekilde saklanır.
- Vector sınıfından türetilmiştir ve genişletilmiş bir versiyonudur.
- Geri alma (undo) işlemleri, metin analizi ve derinlik öncelikli arama gibi senaryolarda kullanılır.
- Stack sınıfı, daha modern ve genel kullanıma uygun alternatiflerle değiştirilmiştir (örneğin, Deque arabirimini uygulayan LinkedList).



# Stack Sınıfı

- Temel Metotlar:
  - `push(E item)`: Stack'e bir eleman ekler.
  - `pop()`: Stack'ten en üstteki elemanı çıkarır ve geri döndürür.
  - `peek()`: Stack'teki en üstteki elemanı döndürür ancak çıkarmaz.
  - `empty()`: Stack'in boş olup olmadığını kontrol eder.
  - `search(Object o)`: Stack içinde belirtilen elemanın konumunu bulur.



# Queue Arayüzü

- FIFO (First-In-First-Out) sıra kuralına göre çalışan bir kuyruğu temsil eder.
- Elemanlar kuyruğa eklendikleri sırayla çıkartılır.
- Elemanlar genellikle offer, poll, ve peek gibi metotlarla işlenir.
- Queue Arayüzü uygulamaları:
  - LinkedList: Standart bir çift yönlü bağlı liste olmasına ek olarak, Queue operasyonlarını da destekler.
  - PriorityQueue: Öncelikli elemanlara göre sıralama yapar.
  - ArrayDeque: Dinamik bir dizi olarak çalışır ve hem kuyruk hem de yığıt işlevselliği sağlar.



# Queue Arayüzü

- Temel Metotlar:
  - offer(E e): Belirtilen öğeyi kuyruğa ekler.
  - poll(): Kuyruktan bir öğe çıkarır ve çıkarılan öğeyi döndürür.
  - peek(): Kuyruktaki ilk öğeyi döndürür ancak kuyruktan çıkartmaz.
  - size(): Kuyruktaki eleman sayısını döndürür.





# PriorityQueue Sınıfı

- Öncelikli kuyruk yapısını uygular ve özellikle sıralama veya öncelikli işlemler gerektiren durumlarda kullanılır.
- Elemanlar, önceliklerine göre küçükten büyüğe veya büyükten küçüğe doğru sıralanabilir.
- Elemanları öncelik sırasına göre depolar, böylece en öncelikli eleman kolayca alınabilir.
- Dinamik olarak büyüyebilir ve performanslıdır.
- Öncelikli kuyruk, elemanların doğru sıralanması için karşılaştırılabilir (comparable) olmalıdır veya bir karşılaştırıcı (comparator) kullanılmalıdır.



# PriorityQueue Sınıfı

- Temel Metotlar:
  - `add(E element)`: PriorityQueue'ye bir eleman ekler.
  - `poll()`: PriorityQueue'den en öncelikli elemanı çıkarır ve döndürür.
  - `peek()`: En öncelikli elemanını döndürür ancak çıkarmaz.
  - `remove(Object obj)`: Belirtilen elemanı PriorityQueue'den kaldırır.
  - `size()`: PriorityQueue'nin boyutunu (eleman sayısını) döndürür.



# Deque Arayüzü

- Deque (Double-Ended Queue) Arayüzü, hem baştan hem de sondan eleman eklenebilen ve çıkarılabilen bir çift yönlü kuyruğu temsil eder.
- İki uçta da eleman eklenebilir ve çıkarılabilir.
- Queue ve Stack işlevselliğini birleştirir.
- Deque Arayüzü uygulamaları:
  - LinkedList: Çift yönlü bağlı liste olarak çalışır ve Deque işlemlerini destekler.
  - ArrayDeque: Dinamik bir dizi olarak çalışır, hem kuyruk hem de yığıt işlevselliği sağlar.



# Deque Arayüzü

- Temel Metotlar:
  - `addFirst(E e)`: Belirtilen öğeyi listenin başına ekler.
  - `addLast(E e)`: Belirtilen öğeyi listenin sonuna ekler.
  - `removeFirst()`: Listenin başındaki elemanı kaldırır ve kaldırılan elemanı döndürür.
  - `removeLast()`: Listenin sonundaki elemanı kaldırır ve kaldırılan elemanı döndürür.
  - `getFirst()`: Listenin başındaki elemanı döndürür ancak kaldırmaz.
  - `getLast()`: Listenin sonundaki elemanı döndürür ancak kaldırmaz.
  - `size()`: Listenin eleman sayısını döndürür.



# ArrayDeque Sınıfı

- Çift uçlu kuyruk yapısını uygular ve hem kuyruk hem de yığın işlemlerini destekler.
- Dinamik bir dizi (array) tabanlı bir veri yapısıdır.
- ArrayDeque sınıfı, genellikle daha hafif ve performanslı alternatiflerle (örneğin, LinkedList) karşılaştırıldığında tercih edilir.

```
ArrayDeque<Integer> sayilar = new ArrayDeque<>();  
sayilar.addFirst(5);  
sayilar.addLast(10);  
int ilkSayi = sayilar.pollFirst();
```



# ArrayDeque Sınıfı

- Temel Metotlar:
  - `addFirst(E element)`: ArrayDeque'nin başına bir eleman ekler.
  - `addLast(E element)`: ArrayDeque'nin sonuna bir eleman ekler.
  - `pollFirst()`: ArrayDeque'nin başındaki elemanı çıkarır ve döndürür.
  - `pollLast()`: ArrayDeque'nin sonundaki elemanı çıkarır ve döndürür.
  - `peekFirst()`: En baştaki elemanı döndürür ancak çıkarmaz.
  - `peekLast()`: Sondaki elemanı döndürür ancak çıkarmaz.



# Set Arayüzü

- Java'da benzersiz elemanları depolamak ve temel küme işlemlerini gerçekleştirmek için kullanılır.
- Elemanlar arasında sıralama garantisi vermez.
- Her eleman yalnızca bir kez bulunabilir.
- Ekleme ve çıkarma işlemleri hızlıdır.
- Set arayüzü uygulamaları:
  - **HashSet**: Benzersiz elemanları depolar, sıralama garantisi vermez.
  - **TreeSet**: Elemanları sıralı bir şekilde depolar, doğal sıralama veya belirtilen bir comparator ile.
  - **LinkedHashSet**: Elemanları eklenme sırasına göre depolar.



# Set Arayüzü

- Temel Metotlar:
  - `add(E e)`: Belirtilen öğeyi kümeye ekler.
  - `remove(Object o)`: Belirtilen öğeyi kümeden kaldırır.
  - `contains(Object o)`: Belirtilen öğenin küme içinde olup olmadığını kontrol eder.
  - `size()`: Kümenin eleman sayısını döndürür.





# HashSet

- Java'da benzersiz elemanları sırasız bir şekilde depolayan ve hashing mantığına dayanan temel bir küme sınıfıdır.
- AbstractSet sınıfından türetilmiştir ve Set arayüzünü uygular.
- Benzersizlik: Her eleman yalnızca bir kez bulunabilir.
- Sırasızlık: Elemanlar eklendikleri sırayla depolanmazlar.
- Elemanların depolanma sırasını garanti etmez.
- Hızlı ekleme ve eleman kontrolü için hashing kullanır.

```
Set<String> meyveler = new HashSet<>();  
meyveler.add("Elma");  
meyveler.add("Armut");
```



# LinkedHashSet Sınıfı

- Elemanların eklenme sırasını koruyan ve sadece tekil elemanları içeren bir küme veri yapısıdır.
- Hızlı erişim sağlar ve benzersiz elemanları garanti eder.
- LinkedHashSet, performans ve sıralama önemli olduğunda tercih edilir; ancak sıralama önemli değilse ve performans kritikse, HashSet kullanılabilir.

```
LinkedHashSet<Integer> sayilar = new LinkedHashSet<>();  
sayilar.add(5);  
sayilar.add(10);
```



# LinkedHashSet Sınıfı

- Temel Metotlar:
  - `add(E element)`: LinkedHashSet'e bir eleman ekler.
  - `remove(Object obj)`: Belirtilen elemanı LinkedHashSet'ten kaldırır.
  - `contains(Object obj)`: Belirtilen elemanın bulunup bulunmadığını döner.
  - `size()`: LinkedHashSet'in boyutunu (eleman sayısını) döndürür.
  - `clear()`: LinkedHashSet'i boşaltır, tüm elemanları kaldırır.



# SortedSet Arayüzü

- Elemanları sıralı bir şekilde depolayan ve benzersiz elemanlardan oluşan bir küme yapısını temsil eder.
- Elemanlar, doğal sıralama veya belirtilen bir comparator'a göre sıralanır.
- Alt küme ve üst küme işlemleri gibi özellikler sunar.
- SortedSet Arayüzü uygulamaları:
  - TreeSet: Elemanları doğal sıralama veya belirtilen bir comparator'a göre sıralı bir şekilde depolar.



# SortedSet Arayüzü

- Temel Metotlar:
  - `add(E e)`: Belirtilen öğeyi küme içine ekler.
  - `remove(Object o)`: Belirtilen öğeyi kümeden kaldırır.
  - `first()`: Kümenin en küçük (ilk) elemanını döndürür.
  - `last()`: Kümenin en büyük (son) elemanını döndürür.
  - `headSet(toElement)`: Kümenin belirtilen elemanına kadar olan alt kümesini döndürür.
  - `tailSet(fromElement)`: Kümenin belirtilen elemanından sonraki üst kümesini döndürür.
  - `subSet(fromElement, toElement)`: Kümenin belirtilen iki eleman arasındaki alt kümesini döndürür.
  - `comparator()`: Kümeyi sıralayan comparator'ı döndürür.



# TreeSet Sınıfı

- Java'da benzersiz elemanları doğal sıralama veya belirtilen bir sıralama mantığına göre depolayan temel bir küme sınıfıdır.
- TreeSet Sınıfı, AbstractSet sınıfından türetilmiştir ve NavigableSet arayüzünü uygular.
- Benzersizlik: Her eleman yalnızca bir kez bulunabilir.
- Sıralılık: Elemanlar doğal sıralama veya belirtilen bir sıralama mantığına göre depolanır.
- Hızlı erişim ve sıralı küme işlevselliği sağlar.

```
TreeSet<String> meyveler = new TreeSet<>();  
meyveler.add("Elma");  
meyveler.add("Armut");
```



# Map Arayüzü

- Java'da anahtar-değer çiftlerini depolayan ve bu çiftler üzerinde işlemler gerçekleştirmeye olanak tanıyan temel bir arayüzdür.
- Veri çiftleri arasında hızlı erişim sağlar.
- Anahtarları ve değerleri birlikte kullanma esnekliği sunar.
- Map Arayüzü uygulamaları:
  - HashMap: Anahtar-değer çiftlerini depolar, sıralama garantisi vermez.
  - TreeMap: Anahtarları sıralı bir şekilde depolar, doğal sıralama veya belirtilen bir comparator ile.
  - LinkedHashMap: Elemanları ekleme sırasına göre depolar.



# Map Arayüzü

- Temel Metotlar:
  - put(K key, V value): Belirtilen anahtarla belirtilen değeri eşleştirir.
  - get(Object key): Belirtilen anahtara karşılık gelen değeri döndürür.
  - remove(Object key): Belirtilen anahtara karşılık gelen değeri kaldırır.
  - containsKey(Object key): Belirtilen anahtarın haritada olup olmadığını kontrol eder.
  - keySet(): Haritadaki tüm anahtarları bir Set olarak döndürür.
  - values(): Haritadaki tüm değerleri bir Collection olarak döndürür.
  - entrySet(): Haritadaki tüm anahtar-değer çiftlerini bir Set olarak döndürür.





# HashMap Sınıfı

- Java'da anahtar-değer çiftlerini depolayan ve hızlı erişim sağlayan temel bir harita sınıfıdır.
- AbstractMap sınıfından türetilmiştir ve Map arayüzünü uygular.
- Benzersiz Anahtarlar: Her anahtar yalnızca bir kez bulunabilir.
- Null Değerler: Birden çok null değerine izin verir.
- Anahtarlar ve değerlerle çalışır. Elemanlara hızlı erişim sağlar.
- Elemanlar sırasız bir şekilde depolanır.

```
HashMap<String, Double> urunFiyatlari = new HashMap<>();  
urunFiyatlari.put("Telefon", 2000.0);
```



# LinkedHashMap Sınıfı

- Elemanların eklenme sırasını koruyan ve benzersiz anahtar-değer çiftlerini içeren bir harita yapısıdır.
- Elemanları eklenme sırasına göre depolar.
- Her bir eleman bir anahtar (key) ve bir değer (value) çiftidir.
- Anahtarlar tekildir, yani her anahtar sadece bir defa bulunabilir.
- LinkedHashMap, performans ve sıralama önemli olduğunda tercih edilir; sıralama önemli değilse ve performans kritikse, HashMap kullanılabilir.

```
LinkedHashMap<String, String> rehber = new LinkedHashMap<>();  
rehber.put("Ahmet", "555-1234");
```



# LinkedHashMap Sınıfı

- Temel Metotlar:
  - put(K key, V value): LinkedHashMap'e bir anahtar-değer çifti ekler.
  - remove(Object key): Belirtilen anahtara sahip çifti map'ten kaldırır.
  - get(Object key): Belirtilen anahtara sahip değeri döndürür.
  - containsKey(Object key): Belirtilen anahtarın LinkedHashMap'te bulunup bulunmadığını kontrol eder.
  - size(): LinkedHashMap'in boyutunu (çift sayısını) döndürür.



# Hashtable Sınıfı

- Anahtar-değer çiftlerini saklamak için kullanılan senkronize bir harita yapısıdır.
- Her bir eleman bir anahtar (key) ve bir değer (value) çiftidir.
- Anahtarlar ve değerler null olamaz.
- Hashtable sınıfı, HashMap gibi senkronize (synchronize) edilmiştir, thread-safe kullanım sağlar.
- Modern alternatiflere göre düşük performanslıdır. Yerine HashMap veya ConcurrentHashMap tercih edilir.



# Hashtable Sınıfı

- Temel Metotlar:
  - put(K key, V value): Hashtable'e bir anahtar-değer çifti ekler.
  - remove(Object key): Belirtilen anahtara sahip çifti kaldırır.
  - get(Object key): Belirtilen anahtara sahip değeri döndürür.
  - containsKey(Object key): Belirtilen anahtarın Hashtable'de bulunup bulunmadığını kontrol eder.
  - size(): Hashtable'in boyutunu (çift sayısını) döndürür.

```
Hashtable<String, String> telefonRehberi = new Hashtable<>();  
telefonRehberi.put("Ahmet", "555-1234");
```



# SortedMap Arayüzü

- SortedMap Arayüzü, anahtar-değer çiftlerini sıralı bir şekilde depolayan bir harita yapısını temsil eder.
- Anahtarlar sıralı bir şekilde depolanır.
- Her anahtar yalnızca bir kez bulunabilir.
- Alt harita ve üst harita gibi özellikler sunar.
- SortedMap Arayüzü uygulamaları:
  - TreeMap: Anahtarları doğal sıralama veya belirtilen bir comparator'a göre sıralı bir şekilde depolar.



# SortedMap Arayüzü

- Temel Metotlar:
  - put(K key, V value): Belirtilen anahtarla belirtilen değeri eşleştirir.
  - get(Object key): Belirtilen anahtara karşılık gelen değeri döndürür.
  - remove(Object key): Belirtilen anahtara karşılık gelen değeri kaldırır.
  - firstKey(): Haritanın en küçük (ilk) anahtarını döndürür.
  - lastKey(): Haritanın en büyük (son) anahtarını döndürür.
  - headMap(toKey): Haritanın belirtilen anahtara kadar olan alt haritasını döndürür.
  - tailMap(fromKey): Haritanın belirtilen anahtardan sonraki üst haritasını döndürür.
  - subMap(fromKey, toKey): Haritanın belirtilen iki anahtar arasındaki alt haritasını döndürür.
  - comparator(): Haritayı sıralayan comparator'ı döndürür.



# TreeMap Sınıfı

- Java'da anahtar-değer çiftlerini doğal sıralama veya belirtilen sıralama mantığına göre depolayan temel bir harita sınıfıdır.
- AbstractMap sınıfından türetilmiştir ve NavigableMap arayüzünü uygular.
- Sıralı Anahtarlar: Anahtarlar doğal sıralama veya belirtilen sıralama mantığına göre depolanır.
- Red-Black Tree: Kırmızı-Siyah ağaç yapısı kullanarak sıralama sağlar.

```
TreeMap<String, Double> urunFiyatlari = new TreeMap<>();  
urunFiyatlari.put("Telefon", 2000.0);
```





# Iterator

- Koleksiyonlarda sıralı bir şekilde gezinmek için kullanılan güçlü bir araçtır.
- Koleksiyonlardaki elemanlara sıralı bir şekilde erişimi sağlar.
- Koleksiyon üzerinde güvenli bir şekilde gezinmeyi sağlar.
- Elemanları okuma, ekleme ve kaldırma işlemlerini destekler.
- Temel Metotlar:
  - `hasNext()`: Bir sonraki elemanın olup olmadığını kontrol eder.
  - `next()`: Iterator üzerinden bir sonraki elemanı alır.
  - `remove()`: Iterator üzerinden son alınan elemanı koleksiyondan kaldırır.



# Iterator

- Iterator genellikle while veya for döngüleri içinde kullanılarak koleksiyon üzerinde iterasyon sağlar.

```
ArrayList<Integer> liste = new ArrayList<>();  
liste.add(3);
```

```
Iterator<Integer> iterator = liste.iterator();  
while (iterator.hasNext()) {  
    Integer sayi = iterator.next();  
    // Eleman üzerinde işlemleri gerçekleştir.  
    System.out.println(sayi);  
}
```

# API Algoritmaları





# API Algoritmaları - Sıralama

- Java Collection API, çeşitli sıralama algoritmalarını ve bu algoritmaları kullanarak kolayca sıralama yapma imkanı sunar.
- Veri sıralama işlemlerinde uygun algoritmanın seçilmesi, performansı önemli ölçüde etkiler.
- Java'nın Collections sınıfındaki sıralama metotları, genel kullanım için hızlı ve etkilidir.



# Temel Sıralama Algoritmaları

- Bubble Sort (Kabarcık Sıralama): İki komşu elemanın karşılaştırıldığı ve gerekli durumda yer değiştirdiği bir algoritmadır. Büyük koleksiyonlarda etkili değildir.
- Selection Sort (Seçmeli Sıralama): Minimum değeri bulup listenin başına yerleştirerek sıralama yapar. Büyük koleksiyonlarda yavaş çalışabilir.
- Insertion Sort (Ekleme Sıralama): Elemanları sırayla alır ve uygun konuma ekler. Küçük koleksiyonlarda etkili, büyük koleksiyonlarda yavaş çalışabilir.
- Merge Sort (Birleştirme Sıralama): Bölme ve birleştirme mantığına dayanır, özyinelemeli bir algoritmadır. Büyük koleksiyonlarda etkili ve kararlıdır.
- Quick Sort (Hızlı Sıralama): Pivot eleman seçimi ve bölme-araştırma stratejisi ile çalışan etkili bir sıralama algoritmasıdır. Ortalama durumlarda hızlı çalışır.



# Java Collection API Sıralama Metotları

- `sort(List<T> list)`:
  - Listenin elemanlarını doğal sıralama (elemanların `Comparable` arayüzünü uygulamış olmalarına göre) kullanarak sıralar.
- `sort(List<T> list, Comparator<? super T> c)`:
  - Listenin elemanlarını belirtilen bir karşılaştırıcı kullanarak sıralar.
- `reverse(List<?> list)`:
  - Belirtilen listenin elemanlarını ters sırayla düzenler.



# Java Collection API Algoritmaları - Karıştırma

- Koleksiyon elemanlarını karıştırmak için kullanılan algoritmalarıdır.
- Karıştırma işlemi, öğelerin rastgele sırayla yer değiştirilmesiyle gerçekleşir.
- Karıştırma algoritmaları genellikle eşit olmayan dağılımlar sağlar, bu nedenle rastgelelik istenen durumlarda kullanılır.
- Collections.shuffle() Metodu: Fisher-Yates Shuffle algoritmasını kullanarak bir listenin elemanlarını karıştırır.
- Knuth tarafından önerilen bu algoritma, listenin elemanlarını rastgele bir sırayla yer değiştirerek karıştırır. Hızlı ve etkilidir.

```
List<String> ogrenciler = new ArrayList<>();  
Collections.shuffle(ogrenciler);
```



# Java Collection API Algoritmaları - Arama

- Java'da koleksiyonlarda eleman aramak için kullanılan algoritmalarıdır.
- Eleman arama işlemlerinde uygun algoritmanın seçilmesi, performans açısından önemlidir.
- İkili arama, sıralı koleksiyonlarda hızlı ve etkili bir arama sağlar.





# Temel Arama Algoritmaları

- Linear Search (Lineer Arama):
  - Elemanları sırayla kontrol ederek aranan elemanı bulmaya çalışır.
  - Küçük koleksiyonlarda etkilidir ancak büyük koleksiyonlarda performans sorunlarına neden olabilir.
- b. Binary Search (İkili Arama):
  - Sıralı bir koleksiyon üzerinde çalışır ve aranan elemanı hızlı bir şekilde bulmayı sağlar.
  - Koleksiyonun sıralı olması gereklidir.
- c. Collections.binarySearch() Metodu:
  - İkili arama algoritmasını kullanarak sıralı bir listede eleman arar.



# Java Collection API Arama Metodu

- `binarySearch(List<? extends Comparable<? super T>> list, T key)` Metodu:
  - Belirtilen anahtarı sıralı bir listede ikili arama yaparak bulmaya çalışır.

```
List<Integer> sayilar = Arrays.asList(1, 3, 5, 7, 9, 11);  
int indeks = Collections.binarySearch(sayilar, 7);
```



# Java Collection API Algoritmaları - Kompozisyon

- Java'da koleksiyonlar üzerinde işlemler gerçekleştirmek için kullanılan temel kompozisyon algoritmalar şunlardır:
- Collections.copy() Metodu:
  - İki koleksiyon arasında elemanları kopyalamak için kullanılır.
  - Kaynak ve hedef koleksiyonların boyutları eşit olmalıdır.
- Collections.fill() Metodu:
  - Belirtilen değerle bir koleksiyonu doldurmak için kullanılır.
- Collections.reverse() Metodu:
  - Bir koleksiyonun elemanlarını tersine çevirmek için kullanılır.
- Collections.swap() Metodu:
  - İki belirtilen indeksteki elemanın yerini değiştirmek için kullanılır.



# Kullanım Örnekleri

```
List<String> isimler = Arrays.asList("Ahmet", "Ayşe", "Cem");  
List<String> hedef = new ArrayList<>(Arrays.asList("X", "Y", "Z"));  
Collections.copy(hedef, isimler);  
Collections.fill(hedef, "Java");  
Collections.reverse(hedef);  
Collections.swap(hedef, 0, 2);
```



SON