



Bölüm 5: Koleksiyonlar

JAVA ile Nesne Yönelimli Programlama



Java Koleksiyonlar

- Veri yönetimi ve işleme konusunda esnek ve etkili çözümler sunar.
- Birden çok öğeyi saklayan ve yöneten veri yapılarıdır.
- Veri saklamak, sıralamak ve işlemek için güçlü araçlar sağlar.
- Performans ve güvenlik açısından optimize edilmiştir.
- Veri yapısı seçerken ihtiyaca uygun koleksiyon türü seçmek önemlidir.



Koleksiyon Türleri

- **List:** Sıralı, indeksli öğeleri içerir. (ArrayList, LinkedList)
- **Set:** Tekil öğeleri içerir. (HashSet, TreeSet)
- **Map:** Anahtar-değer çiftlerini içerir. (HashMap, TreeMap)
- **Queue:** İlk giren, ilk çıkan mantığıyla çalışır. (LinkedList, PriorityQueue)

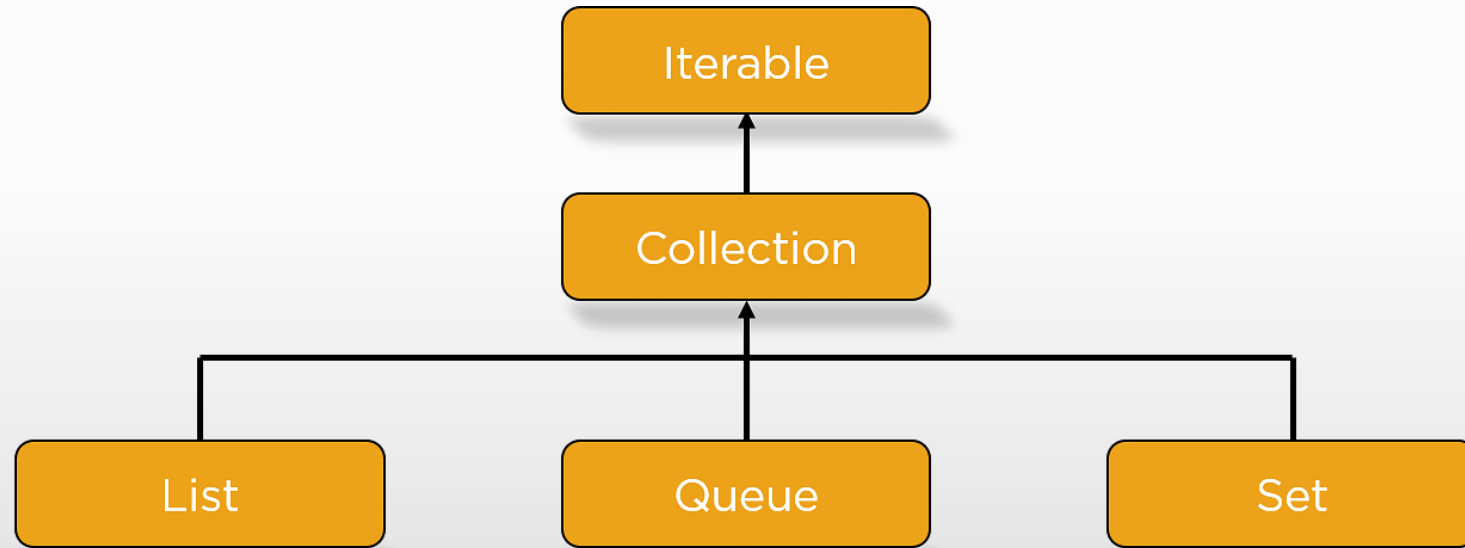


Koleksiyon İşlemleri

- Ekleme, çıkarma, güncelleme, arama gibi temel işlemleri sağlar.
- forEach döngüsüyle kolay *iterasyon* imkanı sağlar.
- Veri hacmi ve erişim ihtiyaçlarına göre optimize edilmiştir.
- Bellek yönetimi ve performans açısından verimli kullanım sağlar.

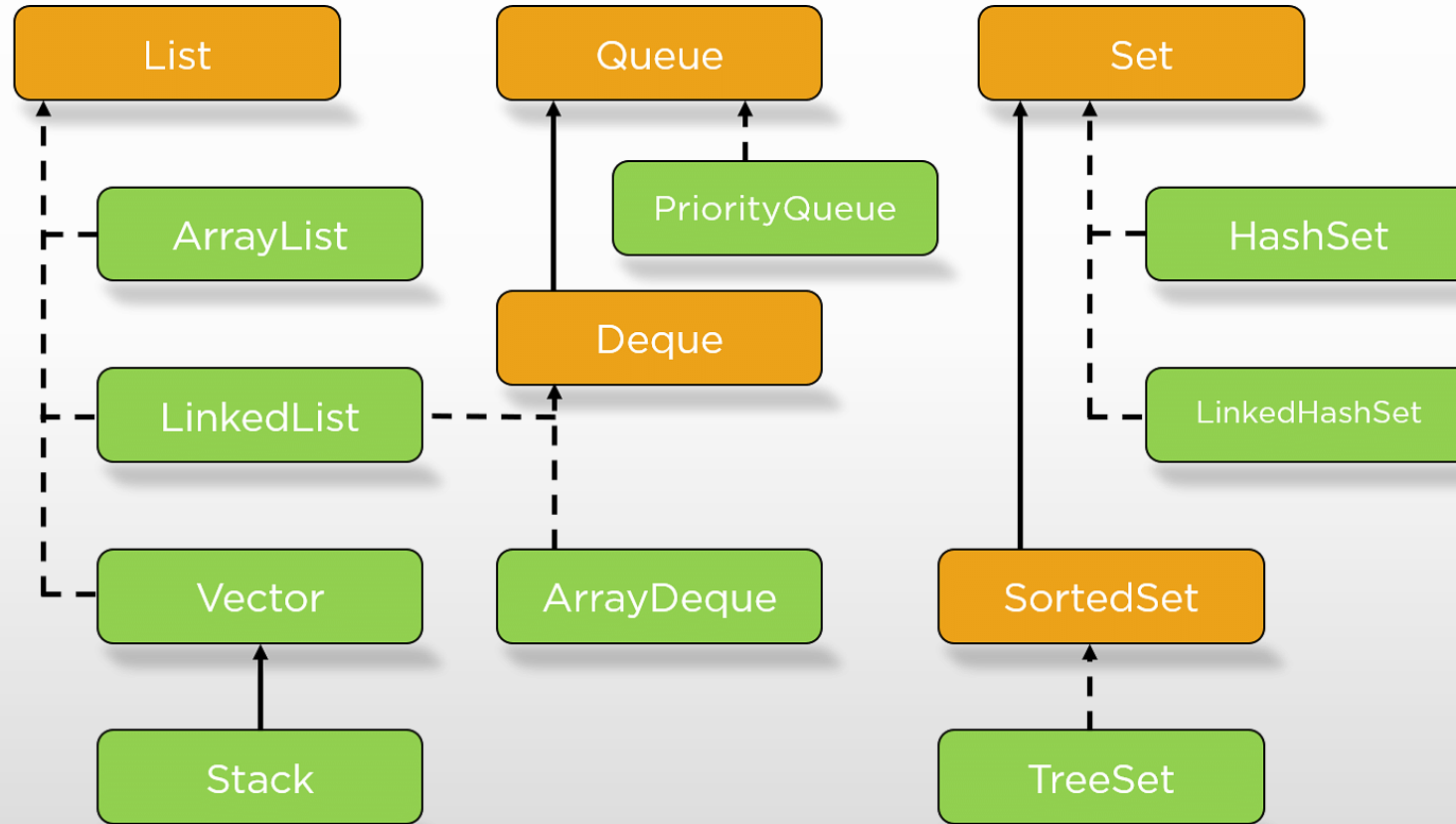


Koleksiyonları Arayüzleri





Koleksiyonları Sınıfları





Koleksiyonlarının Tasarım Amaçları

- Basit ve anlaşılır bir API aracılığıyla **kolay** kullanım sağlar.
- Yüksek performanslı veri saklama ve erişim sağlar.
- *Generic* yapılar kullanılarak, koleksiyonlarda **tip güvenliği** sağlanır.
- Çeşitli veri yapıları ve ihtiyaçlara uygun **geniş sınıf hiyerarşisine** sahiptir.
- Java'nın standart kütüphanesinde bulunan sınıflar ile **uyumludur**.
- **Güvenli** veri erişimi ve manipülasyonu sağlamak üzere tasarlanmıştır.



Koleksiyonlarının Tasarım Amaçları

- API, kodun **anlaşılabilirliğini** artırır ve bakımını kolaylaştırır.
- Koleksiyonlar üzerinde birçok **hazır algoritma** bulunur.
- **Hata durumlarını** ele alacak şekilde tasarlanmıştır.
- 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**, veri yapısının detaylarını gerçekleştirir.
- **Algoritmalar**, işlemleri yürütmek için çeşitli stratejiler sağlar.

- **Modüler tasarım**: Her bileşenin belirli bir rolü ve sorumluluğu vardır.
- **Esneklik**: Farklı uygulama ve algoritmalar, aynı arayüz ile 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 sıralar.



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 liste olarak kullanılabilir.
- Her bir sınıf, belirli bir veri yapısının gereksinimlerini karşılar.
- **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

- Koleksiyon Arayüzü, koleksiyon sınıflarının ortak metotlarını belirler.
- Temel Metotlar:
 - **add(E e)**: öğeyi koleksiyona ekler.
 - **remove(Object o)**: öğeyi koleksiyondan çıkarır.
 - **size()**: koleksiyonun eleman sayısını döndürür.
 - **isEmpty()**: koleksiyonun boş olup olmadığını kontrol eder.
- Döngü Metotları:
 - **iterator()**: koleksiyon üzerinde bir iterator nesnesi 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)**: Öğenin koleksiyonda olup olmadığını kontrol eder.
 - **containsAll(Collection<?> c)**: koleksiyonun tüm elemanlarının aranılan koleksiyon içinde olup olmadığını kontrol eder.
- Temizleme:
 - **clear()**: Koleksiyondaki tüm öğeleri çıkarır.



List Arayüzü

- Öğelerin eklenme zamanına göre sıralı koleksiyonları temsil eder.
- Elemanlar sıralı (ordered not sorted) bir şekilde depolanır.
- İndeks tabanlı erişim sunar.
- **List Arayüzü uygulamaları:**
 - **ArrayList:** Dinamik dizi gibi davranır, hızlı indeks tabanlı erişim sağlar.
 - **LinkedList:** Öğelerin birbirine çift yönlü bağlandığı liste olarak çalışır, ekleme ve çıkarma işlemleri hızlıdır.
 - **Vector:** Thread-safe bir versiyondur, ancak yerine ArrayList tercih edilir.
- Öğe ekleme ve çıkarma işlemlerinin sık olduğu senaryolarda LinkedList tercih edilebilir.



List Arayüzü

- Temel Metotlar:
 - **add(E e)**: öğeyi listenin sonuna ekler.
 - **remove(int index)**: indekstekki öğeyi listeden çıkarır.
 - **get(int index)**: indekstekki öğeyi döndürür.
 - **size()**: eleman sayısını döndürür.



ArrayList Sınıfı

- Dinamik bir dizi yapısı sağlar.
- Elemanları eklenme sırasına göre saklayan temel bir liste sınıfıdır.
- ArrayList sınıfından türetilmiştir ve List arayüzünü uygular.
- **Dinamik Boyut:** Otomatik olarak kendi 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

- Çift yönlü bağlı liste yapısı sağlar.
- Elemanları düğümler aracılığıyla saklayan temel bir liste sınıfıdır.
- AbstractSequentialList sınıfından türetilmiştir ve List arayüzünü uygular.
- **Çift Yönlü Bağlantı:** Düğüm bir önceki ve bir sonraki düğümlerle bağlıdır.
- **Esnek Boyut:** Öğe ekleme ve çıkarma işlemleri için uygundur.

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



Vector Sınıfı

- List arayüzünü uygular.
- Genişletilebilir bir dinamik dizi sağlar.
- İlk oluşturulurken belirtilen kapasiteye sahiptir, elemanlar eklendikçe otomatik olarak genişler.
- Dinamik boyutlandırma özelliği, ve esneklik sağlar.
- Çoklu iş parçacıklı ortamda güvenli bir şekilde kullanılabilir.
- Modern uygulamalarda genellikle daha hafif ve performanslı alternatifler tercih edilir (örneğin, ArrayList).



Vector Sınıfı

- Temel Metotlar:
 - **addElement(E obj):** bir eleman ekler.
 - **elementAt(int index):** indeksteki elemanı döndürür.
 - **removeElement(Object obj):** elemanı çıkarır.
 - **size():** boyutunu (eleman sayısını) döndürür.
 - **capacity():** 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 saklar.
- 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)**: bir eleman ekler.
 - **pop()**: en üstteki elemanı çıkarır ve geri döndürür.
 - **peek()**: en üstteki elemanı döndürür ancak çıkarmaz.
 - **empty()**: boş olup olmadığını kontrol eder.
 - **search(Object o)**: belirtilen elemanın konumunu bulur.



Queue Arayüzü

- FIFO (First-In-First-Out) 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**: çift yönlü bağlı liste yapısındadır.
 - **PriorityQueue**: elemanlar önceliklerine göre sıralanır.
 - **ArrayDeque**: Dinamik bir dizi olarak çalışır.



Queue Arayüzü

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



PriorityQueue Sınıfı

- Öncelikli kuyruk yapısını uygular.
- Sıralama veya öncelikli işlemler gerektiren durumlarda kullanılır.
- Elemanlar, önceliklerine göre küçükten büyüğe veya tam tersi sıralanabilir.
- Elemanları öncelik sırasına göre saklar.
- 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)**: bir eleman ekler.
 - **poll()**: en öncelikli elemanı çıkarır ve döndürür.
 - **peek()**: en öncelikli elemanını döndürür ancak çıkarmaz.
 - **remove(Object obj)**: belirtilen elemanı çıkarır.
 - **size()**: boyutunu (eleman sayısını) döndürür.



Deque Arayüzü

- Deque (Double-Ended Queue) Arayüzü, hem baştan hem de sondan eleman eklenebilen ve çıkarılabilen bir çift yönlü kuyruktur.
- İki uçta da eleman eklenebilir ve çıkarılabilir.
- *Queue* ve *Stack* işlevselliğini birleştirir.
- Deque Arayüzü uygulamaları:
 - **LinkedList**: Çift yönlü bağlı liste olarak çalışır.
 - **ArrayDeque**: Dinamik bir dizi olarak çalışır.



Deque Arayüzü

- Temel Metotlar:
 - **addFirst(E e)**: öğeyi listenin başına ekler.
 - **addLast(E e)**: öğeyi listenin sonuna ekler.
 - **removeFirst()**: baştaki elemanı çıkarır ve döndürür.
 - **removeLast()**: sondaki elemanı çıkarır ve döndürür.
 - **getFirst()**: baştaki elemanı döndürür ancak çıkarmaz.
 - **getLast()**: sondaki elemanı döndürür ancak çıkarmaz.
 - **size()**: eleman sayısını döndürür.



ArrayDeque Sınıfı

- Çift uçlu kuyruk yapısını uygular.
- Hem kuyruk hem yığın işlemlerini destekler.
- Dinamik 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)**: başa bir eleman ekler.
 - **addLast(E element)**: sona bir eleman ekler.
 - **pollFirst()**: baştaki elemanı çıkarır ve döndürür.
 - **pollLast()**: sondaki elemanı çıkarır ve döndürür.
 - **peekFirst()**: baştaki elemanı döndürür ancak çıkarmaz.
 - **peekLast()**: sondaki elemanı döndürür ancak çıkarmaz.



Set Arayüzü

- Tekil elemanları saklama ve küme işlemlerini gerçekleştirmek için kullanılır.
- Elemanlar arasında sıralama garantisi vermez.
- Her elemandan sadece bir tane bulunabilir.
- Ekleme ve çıkarma işlemleri hızlıdır.
- Set arayüzü uygulamaları:
 - **HashSet**: Benzersiz elemanları saklar, sıralama garantisi vermez.
 - **TreeSet**: Elemanları sıralı bir şekilde depolar, doğal sıralama veya belirtilen bir *Comparator* ile.
 - **LinkedHashSet**: Elemanları eklenme sırasına göre saklar.



Set Arayüzü

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



HashSet

- Tekil elemanları sırasız bir şekilde saklar.
- *Hashing* mantığına dayanan temel bir küme sınıfıdır.
- AbstractSet sınıfından türetilmiştir ve Set arayüzünü uygular.
- **Benzersizlik:** Her eleman yalnızca bir tane bulunabilir.
- **Sırasızlık:** Elemanlar eklendikleri sırayla saklanmazlar.
- 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 korur.
- Sadece tekil elemanları içeren bir küme veri yapısıdır.
- Hızlı erişim sağlar ve elemanların benzersizliğini garanti eder.
- LinkedHashSet, performans ve sıralama önemli olduğunda tercih edilir.
- 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)**: bir eleman ekler.
 - **remove(Object obj)**: elemanı çıkarır.
 - **contains(Object obj)**: elemanın bulunup bulunmadığını döner.
 - **size()**: boyutu (eleman sayısını) döndürür.
 - **clear()**: kümeyi boşaltır, tüm elemanları çıkarır.



SortedSet Arayüzü

- Elemanları sıralı bir şekilde saklar.
- Tekil 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)**: öğeyi küme içine ekler.
 - **remove(Object o)**: öğeyi kümeden çıkarır.
 - **first()**: en küçük (ilk) elemanı döndürür.
 - **last()**: en büyük (son) elemanı döndürür.
 - **headSet(to)**: belirtilen öğeye kadar olan alt kümeyi döndürür.
 - **tailSet(from)**: belirtilen öğeden sonraki üst kümeyi döndürür.
 - **subSet(from, to)**: belirtilen iki öğe arasındaki alt kümeyi döndürür.
 - **comparator()**: kümeyi sıralayan comparator'ı döndürür.



TreeSet Sınıfı

- Tekil öğeleri doğal sıralama veya belirtilen sıralama mantığına göre saklar.
- AbstractSet sınıfından türetilmiştir. NavigableSet arayüzünü uygular.
- **Benzersizlik:** Her eleman yalnızca bir kez bulunabilir.
- **Sıralılık:** Elemanlar doğal sıralama veya belirtilen sıralama mantığına göre saklanır.
- Hızlı erişim ve sıralı küme işlevselliği sağlar.

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



Map Arayüzü

- Anahtar-değer çiftlerini saklar.
- Çiftler üzerinde işlemler gerçekleştirmeye olanak tanıyan bir arayüzdür.
- Veri çiftleri arasında hızlı erişim sağlar.
- Anahtarları ve değerleri birlikte kullanma esnekliği sunar.
- Map Arayüzü uygulamaları:
 - **HashMap**: Anahtar-değer çiftlerini saklar, 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 değeri eşleştirir.
 - **get(Object key)**: belirtilen anahtara karşılık gelen değeri döndürür.
 - **remove(Object key)**: belirtilen anahtara karşılık gelen değeri çıkarır.
 - **containsKey(Object key)**: anahtarın eşlemede olup olmadığını kontrol
 - **keySet()**: tüm anahtarları bir *Set* olarak döndürür.
 - **values()**: tüm değerleri bir *Collection* olarak döndürür.
 - **entrySet()**: tüm anahtar-değer çiftlerini bir *Set* olarak döndürür.



HashMap Sınıfı

- Anahtar-değer çiftlerini saklar.
- Hızlı erişim sağlayan temel bir eşleme sınıfıdır.
- *AbstractMap* sınıfından türetilmiştir ve *Map* arayüzünü uygular.
- **Tekil 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 korur.
- Tekil anahtar-değer çiftlerini içeren bir eşleme veri yapısıdır.
- Her bir eleman bir anahtar (key) ve bir değer (value) çiftidir.
- Anahtarlar tekildir, yani her anahtar sadece bir defa bulunabilir.
- 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)**: bir anahtar-değer çifti ekler.
 - **remove(Object key)**: anahtara sahip çifti çıkarır.
 - **get(Object key)**: anahtara sahip değeri döndürür.
 - **containsKey(Object key)**: anahtarın bulunup bulunmadığını kontrol.
 - **size()**: boyutu (çift sayısını) döndürür.



Hashtable Sınıfı

- Anahtar-değer çiftlerini saklamak için kullanılır.
- Senkronize bir eşleme veri yapısıdır.
- Her bir eleman bir anahtar (key) ve bir değer (value) çiftidir.
- Anahtarlar ve değerler *null* olamaz.
- HashMap gibi senkronizedir (synchronize), *thread-safe* kullanım sağlar.
- Modern alternatiflere göre düşük performanslıdır.
- Yerine HashMap veya ConcurrentHashMap tercih edilir.



Hashtable Sınıfı

- Temel Metotlar:
 - **put(K key, V value)**: bir anahtar-değer çifti ekler.
 - **remove(Object key)**: anahtara sahip çifti çıkarır.
 - **get(Object key)**: anahtara sahip değeri döndürür.
 - **containsKey(Object key)**: anahtarın bulunup bulunmadığını kontrol.
 - **size()**: boyutu (çift sayısını) döndürür.

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



SortedMap Arayüzü

- Anahtar-değer çiftlerini sıralı bir şekilde saklar.
- 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)**: anahtarla belirtilen değeri eşleştirir.
 - **get(Object key)**: anahtara karşılık gelen değeri döndürür.
 - **remove(Object key)**: anahtara karşılık gelen değeri çıkarır.
 - **firstKey()**: en küçük (ilk) anahtarı döndürür.
 - **lastKey()**: en büyük (son) anahtarı döndürür.
 - **headMap(to)**: belirtilen anahtara kadar olan alt haritayı döndürür.
 - **tailMap(from)**: belirtilen anahtardan sonraki üst haritayı döndürür.
 - **subMap(from, to)**: belirtilen iki anahtar arasındaki alt haritayı döndürür.
 - **comparator()**: Haritayı sıralayan comparator'ı döndürür.



TreeMap Sınıfı

- Anahtar-değer çiftlerini doğal sıralama veya belirtilen sıralama mantığına göre saklayan temel bir eşleme 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 kullanılır.
- Koleksiyonlardaki elemanlara sıralı bir şekilde erişimi sağlar.
- Koleksiyon üzerinde güvenli bir şekilde gezinmeyi sağlar.
- Elemanları arama, ekleme ve çıkarma 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 çıkarır.



Iterator

- while ve for döngüleri içinde kullanılır.
- Koleksiyon üzerinde gezinme 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 algoritmaları kullanarak kolayca sıralama yapma imkanı sunar.
- Veri sıralama işlemlerinde uygun algoritmanın seçilmesi, performansı önemli ölçüde etkiler.
- *Collections* sınıfındaki sıralama metotları, hızlı ve etkilidir.



Temel Sıralama Algoritmaları

- Bubble Sort (Kabarçık 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 kullanılır.
- Knuth tarafından önerilen bu algoritma, listenin elemanlarını rastgele bir sırayla yer değiştirerek karıştırır. Hızlı ve etkilidir.
- 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.

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



Java Collection API Algoritmaları - Arama

- 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.
- Binary Search (İkili Arama):
 - Sıralı koleksiyon üzerinde çalışır, aranan elemanı hızlı bir şekilde bulur.
 - Koleksiyonun sıralı olması gereklidir.
- 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 kullanılan temel kompozisyon algoritmaları:
- 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