Collections and Maps

Comparing Objects

- The majority of the non-final methods of the Object class are meant to be overridden.
- They provide general contracts for objects, which the classes overriding the methods should honor.

Comparing Objects

- It is important to understand how and why a class should override the equals() and hashCode() methods.
- Implementation of the compareTo() method of the Comparable interface is closely related to the other two methods.

Comparing Objects

- Objects of a class that override the equals() method can be used as elements in a collection.
- If they override the hashCode() method, they can also be used as elements in a HashSet and as keys in a HashMap.
- Implementing the Comparable interface allows them to be used as elements in sorted collections and as keys in sorted maps.
- Next table summarizes the methods that objects should provide if the objects are to be maintained in collections and maps.

Comparing Objects

Concrete Collection/ Map	Interface	Dup- licates	Ordered/ Sorted	Methods That Can Be Called On Elements	Data Structures on Which Implementation Is Based
`HashSet<E>`	`Set<E>`	Unique elements	No order	`equals()` `hashCode()`	Hash table and linked list
`LinkedHash-Set<E>`	`Set<E>`	Unique elements	Insertion order	`equals()` `hashCode()`	Hash table and doubly-linked list
`TreeSet<E>`	`Sorted-Set<E>` `Naviga-bleSet<E>`	Unique elements	Sorted	`compareTo()`	Balanced tree
`ArrayList<E>`	`List<E>`	Allowed	Insertion order	`equals()`	Resizable array

Equivalence Relation

An implementation of the equals() method must satisfy the properties of an equivalence relation:

- Reflexive: For any reference self, self.equals(self) is always true.
- Symmetric: For any references x and y , x.equals(y) is true if and only if y.equals(x) is true.
- Transitive: For any references x, y , and z, if both x.equals(y) and y.equals(z) are true, then x.equals(z) is true.

Equivalence Relation

An implementation of the equals() method must satisfy the properties of an equivalence relation:

- Reflexive: For any reference self, self.equals(self) is always true.
- Symmetric: For any references x and y , x.equals(y) is true if and only if y.equals(x) is true.
- Transitive: For any references x, y , and z, if both x.equals(y) and y.equals(z) are true, then x.equals(z) is true.
- Consistent: For any references x and y , multiple invocations of x.equals(y) will always return the same result, provided the objects referenced by these references have not been modified to affect the equals comparison.
- null comparison: For any non-null reference obj, the call obj.equals(null) always returns false.

Equivalence Relation

- The general contract of the equals() method is defined between objects of arbitrary classes.
- Understanding its criteria is important for providing a proper implementation.

Reflexivity

- This rule simply states that an object is equal to itself, regardless of how it is modified.
- It is easy to satisfy: the object passed as argument and the current object are compared for object reference equality (==):

```
if (this == argumentObj)
   return true;
```

Symmetry

- The expression x.equals(y) invokes the equals() method on the object referenced by the reference x, whereas the expression y.equals(x) invokes the equals() method on the object referenced by the reference y.
- Both invocations must return the same result.

Symmetry

- If the equals() methods invoked are in different classes, the classes must bilaterally agree whether their objects are equal or not.
- In other words, symmetry can be violated if the equals() method of a class makes unilateral decisions about which classes it will interoperate with, while the other classes are not aware of this.
- Avoiding interoperability with other (non-related) classes when implementing the equals() method is strongly recommended.

Transitivity

- If two classes, A and B, have a bilateral agreement on their objects being equal, then this rule guarantees that one of them, say B, does not enter into an agreement with a third class C on its own.
- All classes involved must multilaterally abide by the terms of the contract.

Transitivity

- A typical pitfall resulting in broken transitivity is when the equals() method in a subclass calls the equals() method of its superclass, as part of its equals comparison.
- The equals() method in the subclass usually has code equivalent to the following line:

```
return super.equals(argumentObj) &&
       compareSubclassSpecificAspects();
```

Transitivity

- The idea is to compare only the subclass-specific aspects in the subclass equals() method and to use the superclass equals() method for comparing the superclass-specific aspects.
- However, this approach should be used with extreme caution.
- The problem lies in getting the equivalence contract fulfilled bilaterally between the superclass and the subclass equals() methods.
- If the subclass equals() method does not interoperate with superclass objects, symmetry is easily broken.
- If the subclass equals() method does interoperate with superclass objects, transitivity is easily broken.

Transitivity

- If the superclass is abstract, using the superclass equals() method works well. There are no superclass objects for the subclass equals() method to consider.
- In addition, the superclass equals() method cannot be called directly by any other clients than subclasses.
- The subclass equals() method then has control of how the superclass equals() method is called. It can safely call the superclass equals() method to compare the superclass-specific aspects of subclass objects.

Consistency

- This rule enforces that two objects that are equal (or non-equal) remain equal (or non-equal) as long as they are not modified.
- For mutable objects, the result of the equals comparison can change if one (or both) are modified between method invocations.
- However, for immutable objects, the result must always be the same.
- The equals() method should take into consideration whether the class implements immutable objects, and ensure that the consistency rule is not violated.

null comparison

- This rule states that no object is equal to null. The contract calls for the equals() method to return false. The method must not throw an exception; that would be violating the contract. A check for this rule is necessary in the implementation.
- Typically, the reference value passed as argument is explicitly compared with the null value:

```
if (argumentObj == null)
   return false;
```

null comparison

- In many cases, it is preferable to use the instanceof operator. It always returns false if its left operand is null:

```
if (!(argumentObj instanceof MyRefType))
   return false;
```

- This test has the added advantage that if the condition fails, the argument reference can be safely downcast.

```
public class UsableVNO {
  // Overrides equals(), but not hashCode().
  private int release;
  private int revision;
  private int patch;
  public UsableVNO(int release, int revision, int patch) {
    this.release  = release;
    this.revision = revision;
    this.patch    = patch;
  }
  public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }
  public boolean equals(Object obj) {                    // (1)
    if (obj == this)                                     // (2)
      return true;
    if (!(obj instanceof UsableVNO))                     // (3)
      return false;
    UsableVNO vno = (UsableVNO) obj;                     // (4)
    return vno.patch == this.patch    &&                 // (5)
           vno.revision == this.revision &&
           vno.release  == this.release;
  }
}
```

The hashCode() Method

- Hashing is an efficient technique for storing and retrieving data.
- A common hashing scheme uses an array where each element is a list of items.
- The array elements are called buckets.
- Operations in a hashing scheme involve computing an array index from an item.
- Converting an item to its array index is done by a hash function.
- The array index returned by the hash function is called the hash value of the item.
- The hash value identifies a particular bucket.

The hashCode() Method

Storing an item involves the following steps:

1. Hashing the item to determine the bucket.

2. If the item does not match one already in the bucket, it is stored in the bucket.

Note that no duplicate items are stored. Retrieving an item is based on using a key.

The key represents the identity of the item. Item retrieval is also a two-step process:

1. Hashing the key to determine the bucket.

2. If the key matches an item in the bucket, this item is retrieved from the bucket.

General Contract of the hashCode()

- Consistency during execution: Multiple invocations of the hashCode() method on an object must consistently return the same hash code during the execution of an application, provided the object is not modified to affect the result returned by the equals() method. The hash code need not remain consistent across different executions of the application. This means that using a pseudorandom number generator to produce hash values is not a valid strategy.

- Object value equality implies hash value equality: If two objects are equal according to the equals() method, then the hashCode() method must produce the same hash code for these objects. This tenet ties in with the general contract of the equals() method.

- Object value inequality places no restrictions on the hash value: If two objects are unequal according to the equals() method, then the hashCode() method need not produce distinct hash codes for these objects. It is strongly recommended that the hashCode() method produce unequal hash codes for unequal objects.

The Comparable<E> Interface

- The natural ordering of objects is specified by implementing the generic Comparable Interface. A total ordering of objects can be specified by implementing a comparator that implements the generic Comparator interface.

The Comparable<E> Interface

- The general contract for the Comparable interface is defined by its only method:

```
int compareTo(E o)
```

It returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object, based on the natural ordering.

It throws a ClassCastException if the reference value passed in the argument cannot be compared to the current object.

The Comparable<E> Interface

- Many of the standard classes in the Java API, such as the primitive wrapper classes, String, Date, and File, implement the Comparable interface.
- Objects implementing this interface can be used as elements in a sorted set
- keys in a sorted map
- elements in lists that are sorted manually using the Collections.sort() method

The Comparable<E> Interface

An implementation of the compareTo() method for the objects of a class should meet the following criteria:

- For any two objects of the class, if the first object is less than, equal to, or greater than the second object, then the second object must be greater than, equal to, or less than the first object, respectively, i.e., the comparison is anti-symmetric.
- All three comparison relations (less than, equal to, greater than) embodied in the compareTo() method must be transitive.

For example, if obj1.compareTo(obj2) > 0 and obj2.compareTo(obj3) > 0, then obj1.compareTo(obj3) > 0.

- For any two objects of the class, which compare as equal, the compareTo() method must return the same result if these two objects are compared with any other object, i.e., the comparison is congruent.
- The compareTo() method must be consistent with equals, that is, (obj1.compareTo(obj2) == 0) == (obj1.equals(obj2)).

This is recommended if the objects will be maintained in sorted sets or sorted maps.

The Comparator<E> Interface

All comparators implement the Comparator interface, which has the following single method:

```
int compare(E o1, E o2)
```

- The compare() method returns a negative integer, zero, or a positive integer if the first object is less than, equal to, or greater than the second object, according to the total ordering, i.e., it's contract is equivalent to that of the compareTo() method of the Comparable interface.
- Since this method tests for equality, it is strongly recommended that its implementation does not contradict the semantics of the equals() method.

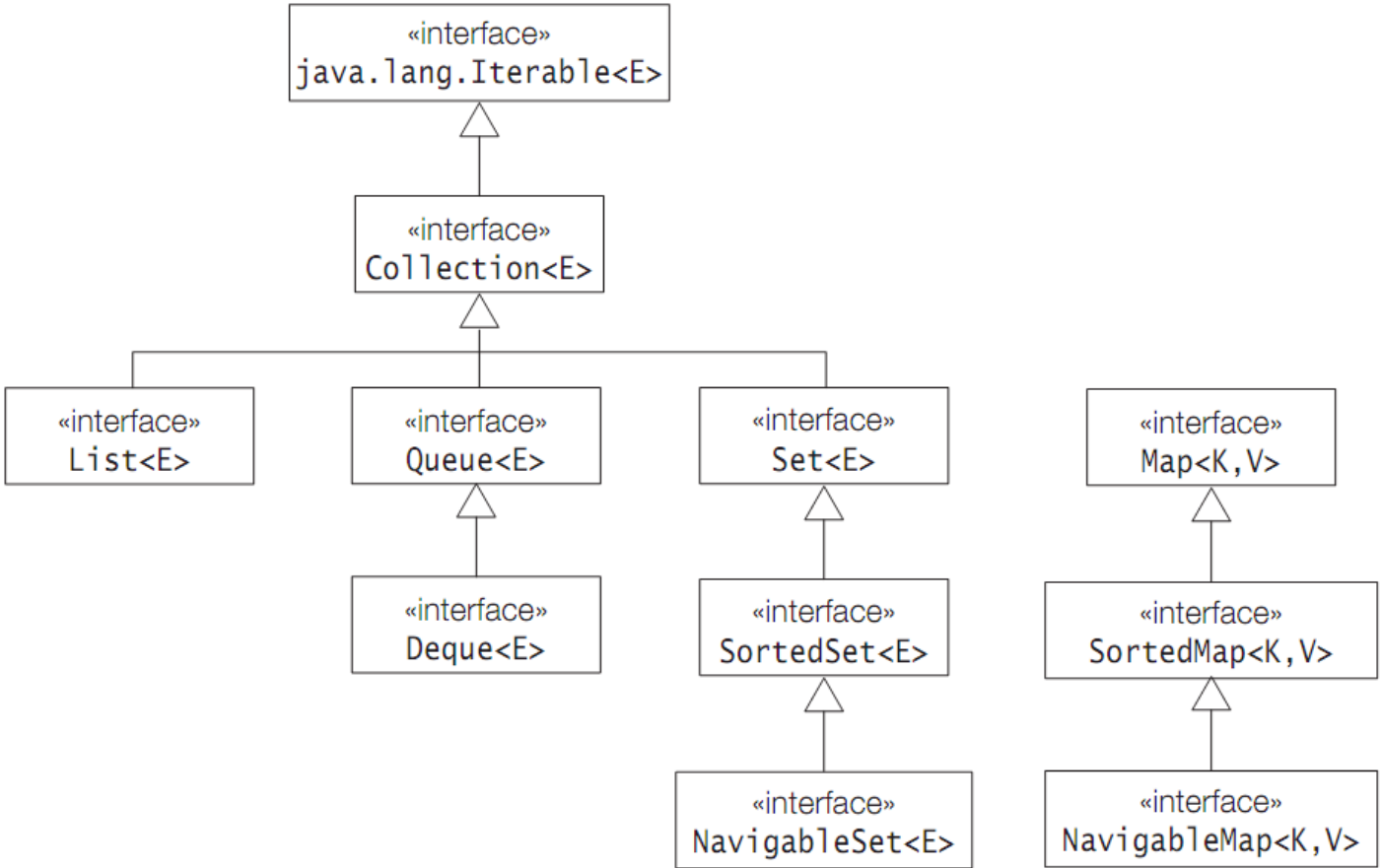
The Java Collections Framework

- A collection allows a group of objects to be treated as a single unit. Objects can be stored, retrieved, and manipulated as elements of a collection.
- Arrays are an example of one kind of collection.
- Program design often requires the handling of collections of objects. The Java Collections Framework provides a set of standard utility classes for managing various kinds of collections.

The Java Collections Framework

- The core framework is provided in the java.util package and comprises three main parts:
- The core interfaces that allow collections to be manipulated independently of their implementation. These generic interfaces define the common functionality exhibited by collections and facilitate data exchange between collections.
- A set of implementations (i.e., concrete classes) that are specific implementations of the core interfaces, providing data structures that a program can readily use.
- An assortment of static utility methods found in the Collections and Arrays classes that can be used to perform various operations on collections and arrays, such as sorting and searching, or creating customized collections.

Core Interfaces



Interface	Description	Concrete Classes
`Collection<E>`	A basic interface that defines the normal operations that allow a collection of objects to be maintained or handled as a single unit.	
`Set<E>`	The `Set` interface extends the `Collection` interface to represent its mathematical namesake: a *set* of unique elements.	`HashSet<E>` `LinkedHashSet<E>`
`SortedSet<E>`	The `SortedSet` interface extends the `Set` interface to provide the required functionality for maintaining a set in which the elements are stored in some sorted order.	`TreeSet<E>`
`NavigableSet<E>`	The `NavigableSet` interface extends and replaces the `SortedSet` interface to maintain a sorted set, and should be the preferred choice in new code.	`TreeSet<E>`

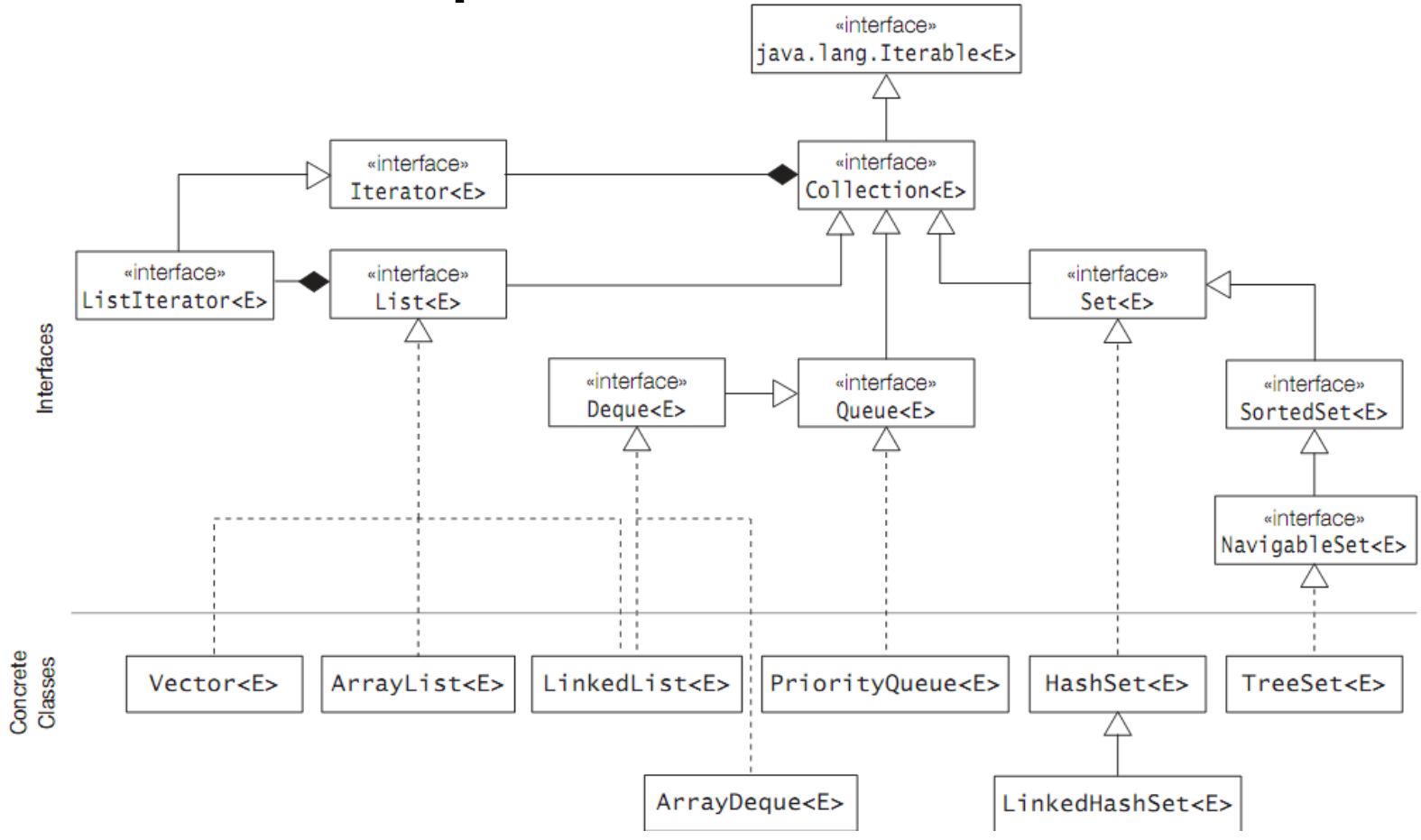
`List<E>`	The `List` interface extends the `Collection` interface to maintain a sequence of elements that can contain duplicates.	`ArrayList<E>`
		`Vector<E>`
		`LinkedList<E>`

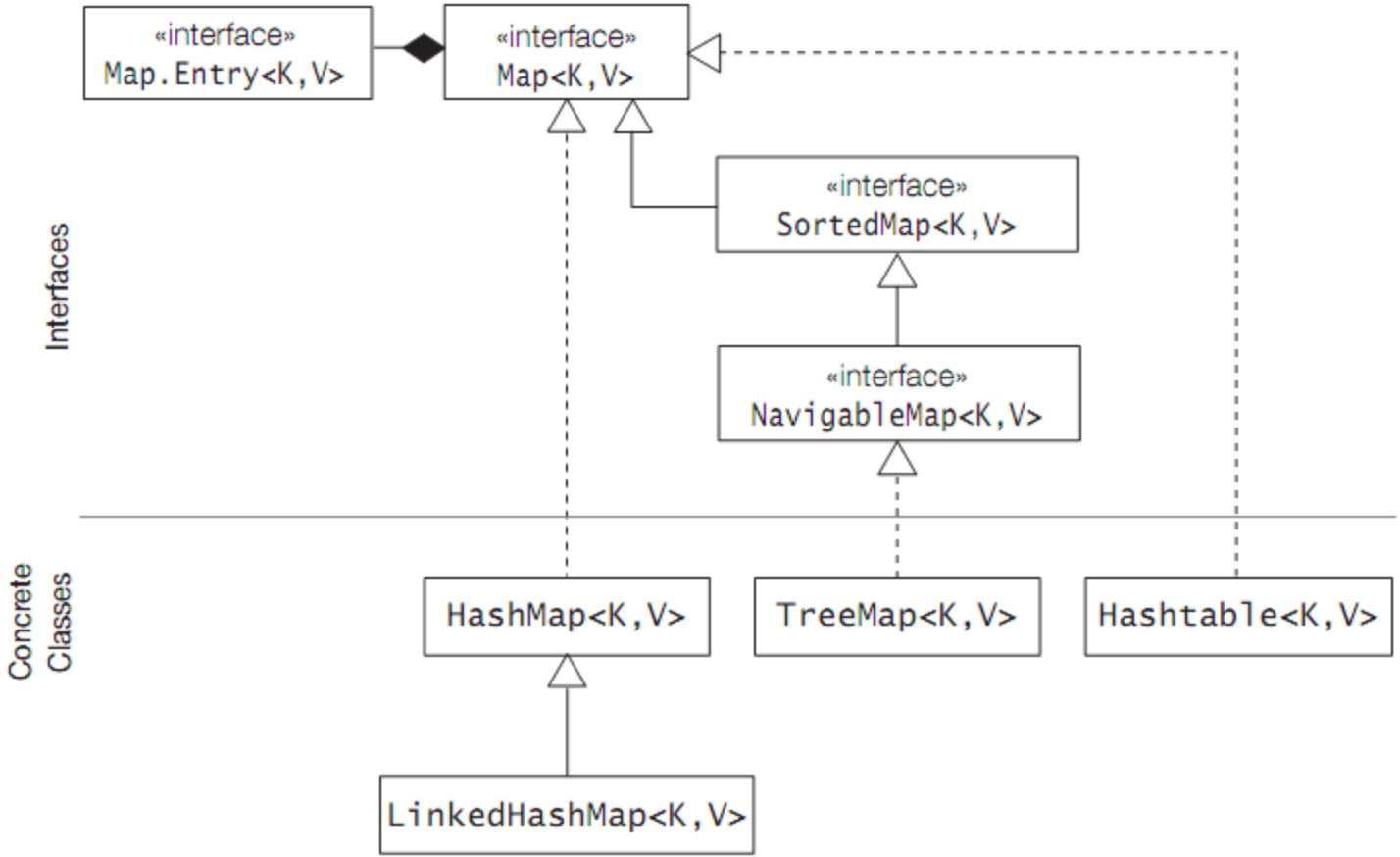
`Queue<E>`	The `Queue` interface extends the `Collection` interface to maintain a collection whose elements need to be processed in some way, i.e., insertion at one end and removal at the other, usually as FIFO (First-In, First-Out).	`PriorityQueue<E>` `LinkedList<E>`
`Deque<E>`	The `Deque` interface extends the `Queue` interface to maintain a queue whose elements can be processed at both ends.	`ArrayDeque<E>` `LinkedList<E>`

`Map<K,V>`	A basic interface that defines operations for maintaining mappings of keys to values.	`HashMap<K,V>`
		`Hashtable<K,V>`
		`LinkedHashMap<K,V>`

`SortedMap<K,V>`	The `SortedMap` interface extends the `Map` interface for maps that maintain their mappings sorted in key order.	`TreeMap<K,V>`
`NavigableMap<K,V>`	The `NavigableMap` interface extends and replaces the `SortedMap` interface for sorted maps.	`TreeMap<K,V>`

Implementations




«interface»
Map.Entry<K,V>
«interface»
Map<K,V>
«interface»
SortedMap<K,V>
«interface»
NavigableMap<K,V>
Interfaces
Concrete
Classes
HashMap<K,V>
TreeMap<K,V>
Hashtable<K,V>
LinkedHashMap<K,V>

Concrete Collection/ Map	Interface	Dup- licates	Ordered/ Sorted	Methods That Can Be Called On Elements	Data Structures on Which Implementation Is Based
`HashSet<E>`	`Set<E>`	Unique elements	No order	`equals()` `hashCode()`	Hash table and linked list
`LinkedHash- Set<E>`	`Set<E>`	Unique elements	Insertion order	`equals()` `hashCode()`	Hash table and doubly-linked list
`TreeSet<E>`	`Sorted- Set<E>` `Naviga- bleSet<E>`	Unique elements	Sorted	`compareTo()`	Balanced tree
`ArrayList<E>`	`List<E>`	Allowed	Insertion order	`equals()`	Resizable array

Concrete Collection/ Map	Interface	Dup- licates	Ordered/ Sorted	Methods That Can Be Called On Elements	Data Structures on Which Implementation Is Based

`LinkedList<E>`	`List<E>`
	`Queue<E>`
	`Deque<E>`
		Allowed	Insertion/ priority/ deque order	`equals()`
				`compareTo()`
					Linked list

`Vector<E>`	`List<E>`	Allowed	Insertion order	`equals()`	Resizable array

Concrete Collection/ Map	Interface	Dup- licates	Ordered/ Sorted	Methods That Can Be Called On Elements	Data Structures on Which Implementation Is Based
`PriorityQueue<E>`	`Queue<E>`	Allowed	Access according to priority order	`compareTo()`	Priority heap (tree-like structure)
`ArrayDeque<E>`	`Queue<E>` `Deque<E>`	Allowed	Access according to either FIFO or LIFO processing order	`equals()`	Resizable array

Concrete Collection/ Map	Interface	Dup-licates	Ordered/ Sorted	Methods That Can Be Called On Elements	Data Structures on Which Implementation Is Based
`HashMap<K,v>`	`Map<K,V>`	Unique keys	No order	`equals()` `hashCode()`	Hash table using array
`LinkedHash-Map<K,V>`	`Map<K,V>`	Unique keys	Key insertion order/ Access order of entries	`equals()` `hashCode()`	Hash table and doubly-linked list
`Hash-table<K,V>`	`Map<K,V>`	Unique keys	No order	`equals()` `hashCode()`	Hash table
`TreeMap<K,V>`	`Sorted-Map<K,V>` `Naviga-bleMap<K,V>`	Unique keys	Sorted in key order	`equals()` `compareTo()`	Balanced tree

Collections - Basic Operations

- The basic operations are used to query a collection about its contents and allow elements to be added to and removed from a collection.

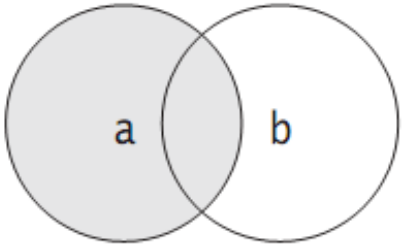
```
int size()
boolean isEmpty()
boolean contains(Object element)
boolean add(E element) // Optional
boolean remove(Object element) // Optional
```

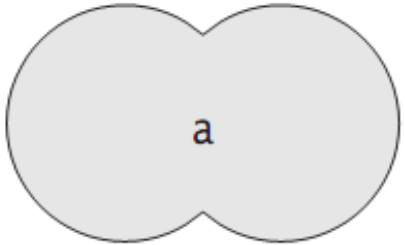
Collections - Bulk Operations

- These operations perform on a collection as a single unit.

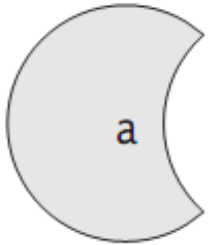
```
boolean containsAll(Collection<?> c)
boolean addAll(Collection<? extends E> c) // Optional
boolean removeAll(Collection<?> c)        // Optional
boolean retainAll(Collection<?> c)        // Optional
void clear()                              // Optional
```

Bulk Operations on Collections

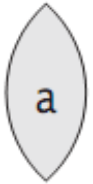




a.addAll(b)



a.removeAll(b)



a.retainAll(b)

Iterators

- A collection provides an iterator which allows sequential access to the elements of a collection.
- An iterator can be obtained by calling the following method of the Collection interface:

```
Iterator<E> iterator()
```

Returns an object which implements the Iterator interface

Iterators

- The generic interface Iterator is defined as follows:

`boolean hasNext()`

Returns true if the underlying collection still has elements left to return. A future call to the next() method will return the next element from the collection.

`E next()`

Moves the iterator to the next element in the underlying collection, and returns the current element. If there are no more elements left to return, it throws a NoSuchElementException.

`void remove() Optional`

Removes the element that was returned by the last call to the next() method from the underlying collection. Invoking this method results in an IllegalStateException if the next() method has not yet been called or when the remove() method has already been called after the last call to the next() method. This method is optional for an iterator, i.e., it throws an UnsupportedOperationException if the remove operation is not supported.

Array Operations

- These operations convert collections to arrays.

```
Object[] toArray()
<T> T[] toArray(T[] a)
```

The first toArray() method returns an array of type Object filled with all the elements of the collection. The second method is a generic method that stores the elements of a collection in an array of type T.

Array Operations

- If the given array is big enough, the elements are stored in this array. If there is room to spare in the array, that is, the length of the array is greater than the number of elements in the collection, the spare room is filled with null values before the array is returned.
- If the array is too small, a new array of type T and appropriate size is created.
- If T is not a supertype of the runtime type of every element in the collection, an ArrayStoreException is thrown.

Sets - Bulk Operations and Set Logic

Set Methods (a and b are sets)	Corresponding Mathematical Operations
`a.containsAll(b)`	`b` $\subseteq$ `a` (subset)
`a.addAll(b)`	`a = a` $\cup$ `b` (union)
`a.removeAll(b)`	`a = a - b` (difference)
`a.retainAll(b)`	`a = a` $\cap$ `b` (intersection)
`a.clear()`	`a` = $\emptyset$ (empty set)

HashSet<E> and LinkedHashSet<E>

- A HashSet relies on the implementation of the hashCode() and equals() methods of its elements.
- The hashCode() method is used for hashing the elements, and the equals() method is needed for comparing elements.
- In fact, the equality and the hash codes of HashSets are defined in terms of the equality and the hash codes of their elements.

HashSet<E> and LinkedHashSet<E>

- `HashSet()`

Constructs a new, empty set.

- `HashSet(Collection c)`

Constructs a new set containing the elements in the specified collection. The new set will not contain any duplicates. This offers a convenient way to remove duplicates from a collection.

- `HashSet(int initialCapacity)`

Constructs a new, empty set with the specified initial capacity.

- `HashSet(int initialCapacity, float loadFactor)`

Constructs a new, empty set with the specified initial capacity and the specified load factor.

SortedSet<E> and NavigableSet<E>

- The SortedSet interface extends the Set interface to provide the functionality for handling sorted sets.
- Since the elements are sorted, traversing the set either using the for(:) loop or an iterator will access the elements according to the ordering used by the set.

SortedSet<E> and NavigableSet<E>

- // First-last elements

```
E first()
E last()
```

- The first() method returns the first element currently in this sorted set, and the last() method returns the last element currently in this sorted set. The elements are chosen based on the ordering used by the sorted set. Both throw a NoSuchElementException if the sorted set is empty.

SortedSet<E> and NavigableSet<E>

- // Range-view operations

```
SortedSet<E> headSet(<E> toElement)
SortedSet<E> tailSet(<E> fromElement)
SortedSet<E> subSet(<E> fromElement, <E> toElement)
```

- The headSet() method returns a view of a portion of this sorted set, whose elements are strictly less than the specified element.
- Similarly, the tailSet() method returns a view of the portion of this sorted set, whose elements are greater than or equal to the specified element.
- The subSet() method returns a view of the portion of this sorted set, whose elements range from fromElement, inclusive, to toElement, exclusive (also called half-open interval). It throws an IllegalArgumentException if the fromElement is greater than the toElement

SortedSet<E> and NavigableSet<E>

- // Comparator access

```
Comparator<? super E> comparator()
```

- This method returns the comparator associated with this sorted set, or null if it uses the natural ordering of its elements.
- This comparator, if defined, is used by default when a sorted set is constructed and also used when copying elements into new sorted sets.

The NavigableSet<E> Interface

- The NavigableSet interface extends the SortedSet interface with navigation methods to find the closest matches for specific search targets.
- In the absence of elements, these operations return null rather than throw a NoSuchElementException.
- The NavigableSet interface replaces the SortedSet interface and is the preferred choice when a sorted set is required.

The NavigableSet<E> Interface

- // First-last elements

```
E pollFirst()
E pollLast()
```

- The pollFirst() method removes and returns the first element and the pollLast() method removes and returns the last element currently in this navigable set.
- The element is determined according to some policy employed by the set—for example, queue policy. Both return null if the sorted set is empty

The NavigableSet<E> Interface

- // Range-view operations

```
NavigableSet<E> headSet(<E> toElement,   boolean inclusive)
NavigableSet<E> tailSet(<E> fromElement, boolean inclusive)
NavigableSet<E> subSet(<E> fromElement, boolean fromInclusive,
      <E> toElement,   boolean toInclusive)
```

- These operations are analogous to the ones in the SortedSet interface returning different views of the underlying navigable set, depending on the bound elements.
- However, the bound elements can be excluded or included by the operation, depending on the value of the boolean argument inclusive.

The NavigableSet<E> Interface

```
// Closest-matches
E ceiling(E e)
E floor(E e)
E higher(E e)
E lower(E e)
```

- The method ceiling() returns the least element in the navigable set greater than or equal to argument e.
- The method floor() returns the greatest element in the navigable set less than or equal to argument e.
- The method higher() returns the least element in the navigable set strictly greater than argument e.
- The method lower() returns the greatest element in the navigable set strictly less than argument e.
- All methods return null if the required element is not found.

The NavigableSet<E> Interface

```
// Reverse order
Iterator<E> descendingIterator()
NavigableSet<E> descendingSet()
```

- The first method returns a reverse-order view of the elements in the navigable set.
- The second method returns a reverse-order iterator for the navigable set.

The TreeSet<E> Class

- The TreeSet implementation uses balanced trees, which deliver excellent (logarithmic) performance for all operations.
- However, searching in a HashSet can be faster than in a TreeSet because hashing algorithms usually offer better performance than the search algorithms for balanced trees.
- The TreeSet class is preferred if elements are to be maintained in sorted order and fast insertion and retrieval of individual elements is desired.

The TreeSet<E> Class

`TreeSet()`

- The default constructor creates a new empty sorted set, according to the natural ordering of the elements.

`TreeSet(Comparator<? super E> comparator)`

- A constructor that takes an explicit comparator for specific total ordering of the elements.

`TreeSet(Collection<? extends E> collection)`

- A constructor that creates a sorted set based on a collection, according to the natural ordering of the elements.

`TreeSet(SortedSet<E> set)`

- A constructor that creates a new set containing the same elements as the specified sorted set, with the same ordering.

```
public class SetNavigation {
  public static void main(String[] args) {
    NavigableSet<String> strSetA = new TreeSet<String>();                    // (1)
    Collections.addAll(strSetA, "Strictly", "Java", "dancing", "ballroom");  // (2)
    out.println("Before: " + strSetA);       // [Java, Strictly, ballroom, dancing]
    out.println("\nSubset-views:");          // (3)
    out.println(strSetA.headSet("ballroom", true));  // [Java, Strictly, ballroom]
    out.println(strSetA.headSet("ballroom", false)); // [Java, Strictly]
    out.println(strSetA.tailSet("Strictly", true));// [Strictly, ballroom, dancing]
    out.println(strSetA.tailSet("Strictly", false)); // [ballroom, dancing]
    out.println(strSetA.subSet("A", false, "Z", false )); // [Java, Strictly]
    out.println(strSetA.subSet("a", false, "z", false )); // [ballroom, dancing]
    out.println("\nClosest-matches:");         // (4)
    out.println(strSetA.ceiling("ball"));      // ballroom
    out.println(strSetA.floor("ball"));        // Strictly
    out.println(strSetA.higher("ballroom"));         // dancing
    out.println(strSetA.lower("ballroom"));          // Strictly
    out.println("\nReverse order:");                 // (5)
    out.println(strSetA.descendingSet());    // [dancing, ballroom, Strictly, Java]
    out.println("\nFirst-last elements:");          // (6)
    out.println(strSetA.pollFirst());        // Java
    out.println(strSetA.pollLast());         // dancing
    out.println("\nAfter: " + strSetA);      // [Strictly, ballroom]
  }
}
```

Lists

```
E get(int index)
E set(int index, E element)   //Optional
void add(int index, E element)//Optional
boolean addAll(
    int index, Collection<? extends E> c)
    //Optional
E remove(int index) Optional
int indexOf(Object o)
int lastIndexOf(Object o)
List<E> subList(int fromIndex, int toIndex)
```

List iterators

```
// List Iterators
ListIterator<E> listIterator()
ListIterator<E> listIterator(int index)
```

The iterator from the first method traverses the elements consecutively, starting with the first element of the list, whereas the iterator from the second method starts traversing the list from the element indicated by the specified index.

List Iterator

```
interface ListIterator<E> extends Iterator<E> {
  boolean hasNext();
  boolean hasPrevious();
  E next();                 // Element after the cursor
  E previous();             // Element before the cursor
  int nextIndex();          // Index of element after the cursor
  int previousIndex();      // Index of element before the cursor
  void remove();            // Optional
  void set(E o);            // Optional
  void add(E o);            // Optional
}
```

ArrayList<E>, LinkedList<E>, Vector<E>

- The **ArrayList** class implements the **List** interface.
- The **Vector** class is a legacy class that has been retrofitted to implement the **List** interface, and will not be discussed in detail.
- The **Vector** and **ArrayList** classes are implemented using dynamically resizable arrays, providing fast random access (i.e., position-based access) and fast list traversal—very much like using an ordinary array.
- Unlike the **ArrayList** class, the **Vector** class is thread-safe, meaning that concurrent calls to the vector will not compromise its integrity.
- The **LinkedList** implementation uses a doubly-linked list.
- Insertions and deletions in a doubly-linked list are very efficient.

ArrayList<E> & LinkedList<E>

- The ArrayList class provides the following constructors:

ArrayList()

ArrayList(Collection<? extends E> c)

- The default constructor creates a new, empty ArrayList.
- The second constructor creates a new ArrayList containing the elements in the specified collection. The new ArrayList will retain any duplicates. The ordering in the ArrayList will be determined by the traversal order of the iterator for the collection passed as argument.
- The LinkedList class provides constructors that are analogous to these two

ArrayList constructors.

ArrayList(int initialCapacity)

- The third constructor creates a new, empty ArrayList with the specified initial capacity.

Queues

- The Queue interface extends the Collection interface with the following methods:

```
boolean add(E element)
boolean offer(E element)
```

- Both methods insert the specified element in the queue. The return value indicates the success or failure of the operation. The add() method inherited from the Collection interface throws an IllegalStateException if the queue is full, but the offer() method does not.

```
E poll()
E remove()
```

- Both methods retrieve the head element and remove it from the queue. If the queue is empty, the poll() method returns null, but the remove() method throws a NoSuchElementException.

```
E peek()
E element()
```

Both methods retrieve the head element, but do not remove it from the queue. If the queue is empty, the peek() method returns null, but the element() method throws a NoSuchElementException.

The PriorityQueue<E> and LinkedList<E> Classes

- As the name suggests, the PriorityQueue class is the obvious implementation for a queue with priority ordering. The implementation is based on a priority heap, a tree-like structure that yields an element at the head of the queue according to the priority ordering, which is defined either by the natural ordering of its elements or by a comparator. In the case of several elements having the same priority, one of them is chosen arbitrarily.
- Elements of a PriorityQueue are not sorted. The queue only guarantees that elements can be removed in priority order, and any traversal using an iterator does not guarantee to abide by the priority order.

PriorityQueue<E>

`PriorityQueue()`

`PriorityQueue(Collection<? extends E> c)`

- The default constructor creates a new, empty PriorityQueue with default initial capacity and natural ordering. The second constructor creates a new PriorityQueue containing the elements in the specified collection. It will have natural ordering of its elements, unless the specified collection is either a SortedSet or another PriorityQueue, in which case, the collection's ordering will be used.

`PriorityQueue(int initialCapacity)`

`PriorityQueue(int initialCapacity, Comparator<? super E> comparator)`

The first constructor creates a new, empty PriorityQueue with the specified initial capacity and natural ordering. The second constructor creates a new, empty PriorityQueue with the specified initial capacity, but the ordering is defined by the specified comparator.

`PriorityQueue(PriorityQueue<? extends E> pq)`

`PriorityQueue(SortedSet<? extends E> set)`

The constructors create a new PriorityQueue with the ordering and the elements from the specified priority queue or sorted set, respectively.

The Deque<E> Interface

- The Deque interface extends the Queue interface to allow double-ended queues. Such a queue is called a deque.
- It allows operations not just at its head, but also at its tail. It is a linear unbounded structure in which elements can be inserted at or removed from either end.
- Various synonyms are used in the literature for the head and tail of a deque: front and back, first and last, start and end.

The Deque<E> Interface

- The Deque interface defines symmetrical operations at its head and tail. Which end is in question is made evident by the method name. Below, equivalent methods from the Queue are also identified.
- The push() and pop() methods are convenient for implementing stacks.

The Deque<E> Interface

```
// Insert
boolean offerFirst(E element)
boolean offerLast(E element)              //Queue equivalent: offer()
void push(E element) Synonym: addFirst()
void addFirst(E element)
void addLast(E element)                   //Queue equivalent: add()
// Remove
E pollFirst() Queue equivalent: poll()
E pollLast()
E pop() Synonym: removeFirst()
E removeFirst()                           //Queue equivalent: remove()
E removeLast()
boolean removeFirstOccurence(Object obj)
boolean removeLastOccurence(Object obj)
// Examine
E peekFirst()                             //Queue equivalent: peek()
E peekLast()
E getFirst()                              //Queue equivalent: element()
E getLast()
// Misc.
Iterator<E> descendingIterator()
```

ArrayDeque<E> & LinkedList<E>

- The ArrayDeque and LinkedList classes implement the Deque interface. The ArrayDeque class provides better performance than the LinkedList class for implementing FIFO queues, and is also a better choice than the java.util.Stack class for implementing stacks.
- An ArrayDeque is also Iterable, and traversal is always from the head to the tail.
- The class provides the descendingIterator() method for iterating in reverse order.
- Since deques are not lists, positional access is not possible, nor can they be sorted.

Maps

- A Map defines mappings from keys to values.
- The <key, value> pair is called an entry.
- A map does not allow duplicate keys, in other words, the keys are unique. Each key maps to one value at the most, implementing what is called a single-valued map.
- Thus, there is a many-to-one relation between keys and values. For example, in a student-grade map, a grade (value) can be awarded to many students (keys), but each student has only one grade.

Maps

- A map is not a collection and the Map interface does not extend the Collection interface.
- However, the mappings can be viewed as a collection in various ways: a key set, a value collection, or an entry set.
- These collection views are the only means of traversing a map.

Maps - Basic Operations

```
Object put(K key, V value)              // Optional
Object get(Object key)
Object remove(Object key)               // Optional
```

- The put() method inserts the <key , value> entry into the map. It returns the old value previously associated with the specified key, if any. Otherwise, it returns the null value.
- The get() method returns the value to which the specified key is mapped, or null if no entry is found.
- The remove() method deletes the entry for the specified key. It returns the value previously associated with the specified key, if any. Otherwise, it returns the null value.

Maps - Basic Operations

```
boolean containsKey(Object key)
boolean containsValue(Object value)
```

- The containsKey() method returns true if the specified key is mapped to a value in the map.
- The containsValue() method returns true if there exists one or more keys that are mapped to the specified value.

```
int size()
boolean isEmpty()
```

- These methods return the number of entries (i.e., number of unique keys in the map) and whether the map is empty or not.

Bulk Operations

- Bulk operations can be performed on an entire map.

```
void putAll(Map<? extends K, ? extends V> map) //Optional
void clear()                                   //Optional
```

- The first method copies all entries from the specified map to the current map, and the second method deletes all entries from the current map.

Collection Views

- Views allow information in a map to be represented as collections.

```
Set<K> keySet()
Collection<V> values()
Set<Map, Entry<K, V>> entrySet()
```

- These methods provide different views of a map. Changes in the map are reflected in the view, and vice versa. These methods return a set view of keys,a collection view of values, and a set view of <key, value> entries, respectively.

Entry<K,V> Interface

Each <key, value> in the entry set view is represented by an object implementing the nested Map.Entry interface. An entry in the entry set view can be manipulated by methods defined in this interface, which are selfexplanatory:

```
interface Entry<K, V> {
    K getKey();
    V getValue();
    V setValue(V value);
}
```

Map Implementations

- HashMap<K,V>
- LinkedHashMap<K,V>
- Hashtable<K,V>

Map Implementations

- The classes HashMap and Hashtable implement unordered maps.
- The class TreeMap implements sorted maps.
- While the HashMap class is not thread-safe and permits one null key, the Hashtable class is thread-safe and permits non-null keys and values only. The thread-safety provided by the Hashtable class comes with a performance penalty.
- Thread-safe use of maps is also provided by the methods in the Collections class.
- Like the Vector class, the Hashtable class is also a legacy class that has been retrofitted to implement the Map interface.

Map Implementations

- The LinkedHashMap implementation is a subclass of the HashMap class. The relationship between the map classes LinkedHashMap and HashMap is analogous to the relationship between their counterpart set classes LinkedHashSet and HashSet.
- Both the HashMap and the LinkedHashMap classes provide comparable performance, but the HashMap class is the natural choice if ordering is not an issue.
- Operations such as adding, removing, or finding an entry based on a key are in constant time, as these hash the key. Operations such as finding the entry with a particular value are in linear time, as these involve searching through the entries.

Map Implementations

- Adding, removing, and finding entries in a LinkedHashMap can be slightly slower than in a HashMap, as an ordered doubly-linked list has to be maintained.
- Traversal of a map is through one of its collection-views. For an underlying LinkedHashMap, the traversal time is proportional to the size of the map—regardless of its capacity.
- However, for an underlying HashMap, it is proportional to the capacity of the map.

The HashMap class

```
HashMap()
HashMap(int initialCapacity)
HashMap(int initialCapacity, float loadFactor)
```

- Constructs a new, empty HashMap, using either specified or default initial capacity and load factor.

```
HashMap(Map<? extends K,? extends V> otherMap)
```

- Constructs a new map containing the elements in the specified map.

The LinkedHashMap class

- In addition, the LinkedHashMap class provides a constructor where the ordering mode can also be specified:

```
LinkedHashMap(int initialCapacity, float loadFactor,
              boolean accessOrder)
```

- Constructs a new, empty LinkedHashMap with the specified initial capacity, the specified load factor, and the specified ordering mode.
- The ordering mode is true for access order and false for key insertion order.

The SortedMap<K,V> and NavigableMap<K,V> Interfaces

- The SortedMap and NavigableMap interfaces are the analogs of the SortedSet and the NavigableSet interfaces, respectively.

The SortedMap<K,V> Interface

- The SortedMap interface extends the Map interface to provide the functionality for implementing maps with sorted keys. Its operations are analogous to those of the SortedSet interface, applied to maps and keys rather than to sets and elements.

```
// First-last keys
K firstKey()                                    //Sorted set: first()
K lastKey()                                     //Sorted set: last()
// Range-view operations
SortedMap<K,V> headMap(K toKey)                 //Sorted set: headSet()
SortedMap<K,V> tailMap(K fromKey)               //Sorted set: tailSet()
SortedMap<K,V> subMap(K fromKey, K toKey)       //Sorted set: sub
// Comparator access
Comparator<? super K> comparator()
```

The NavigableMap<K,V> Interface

- In addition to the methods of the SortedMap interface, the NavigableMap interface adds the new methods shown below, where the analogous methods from the NavigableSet interface are also identified.
- Note that where a NavigableMap method returns a Map.Entry object representing a mapping, the corresponding NavigableSet method returns an element of the set.

The NavigableMap<K,V> Interface

```
// First-last elements
Map.Entry<K, V> pollFirstEntry()    //Navigable set: pollFirst()
Map.Entry<K, V> pollLastEntry()     //Navigable set: pollLast()
Map.Entry<K, V> firstEntry()
Map.Entry<K, V> lastEntry()
// Range-view operations
NavigableMap<K, V> headMap(K toElement,   //Navigable set: headSet()
                     boolean inclusive)
NavigableMap<K, V> tailMap(K fromElement, //Navigable set: tailSet()
                     boolean inclusive)
NavigableMap<K, V> subMap(K fromElement,  //Navigable set: subSet()
      boolean fromInclusive,
            K toElement,
      boolean toInclusive)
```

The NavigableMap<K,V> Interface

```
// Closest-matches
Map.Entry<K, V> ceilingEntry(K key)   //Navigable set: ceiling()
K               ceilingKey(K key)
Map.Entry<K, V> floorEntry(K key)     //Navigable set: floor()
K               floorKey(K key)
Map.Entry<K, V> higherEntry(K key)    //Navigable set: higher()
K               higherKey(K key)
Map.Entry<K, V> lowerEntry(K key)     //Navigable set: lower()
K               lowerKey(K key)
// Navigation
NavigableMap<K, V> descendingMap()    //Navigable set: descendingSet()
NavigableSet<K> descendingKeySet()
NavigableSet<K> navigableKeySet()
```

The TreeMap<K,V> Class

- The TreeMap class provides four constructors, analogous to the ones in the TreeSet class:

```
TreeMap()
```

- A standard constructor used to create a new empty sorted map, according to the natural ordering of the keys.

```
TreeMap(Comparator<? super K> c)
```

- A constructor that takes an explicit comparator for the keys, that is used to order the entries in the map.

```
TreeMap(Map<? extends K, ? extends V> m)
```

- A constructor that can create a sorted map based on a map, according to the natural ordering of the keys.

```
TreeMap(SortedMap<K, ? extends V> m)
```

- A constructor that creates a new map containing the same entries as the specified sorted map, with the same ordering for the keys.

Working with Collections

- The Java Collections Framework also contains two classes, Collections and Arrays, that provide various operations on collections and arrays, such as sorting and searching, or creating customized collections.
- The methods provided are all public and static, therefore these two keywords will be omitted in their method header declarations in this section.
- The methods also throw a NullPointerException if the specified collection or array references passed to them are null.

Ordering Elements in Lists

- The Collections class provides two static methods for sorting lists.

```
<E extends Comparable<? super E>>
void sort(List<E> list)

<E> void sort(List<E> list,
Comparator<? super E> c)
```

Ordering Elements in Lists

```
<E> Comparator<E> reverseOrder()
<E> Comparator<E>
reverseOrder(Comparator<E> comparator)
```

- The first method returns a comparator that enforces the reverse of the natural ordering. The second one reverses the total ordering defined by the comparator. Both are useful for maintaining objects in reverse-natural or reverse-total ordering in sorted collections and arrays.

Utility methods

- The following utility methods apply to any list, regardless of whether the elements are Comparable or not:

void reverse(List<?> list)

- Reverses the order of the elements in the list.

void rotate(List<?> list, int distance)

- Rotates the elements towards the end of the list by the specified distance. A negative value for the distance will rotate toward the start of the list.

void shuffle(List<?> list)

void shuffle(List<?> list, Random rnd)

- Randomly permutes the list, that is, shuffles the elements.

void swap(List<?> list, int i, int j)

- Swaps the elements at indices i and j.

Searching in Collections

```
<E> int binarySearch(List<? extends Comparable<? super E>> list,
                     E key)
<E> int binarySearch(List<? extends E> list, E key,
                     Comparator<? super E> c))
```

- The methods use a binary search to find the index of the key element in the specified sorted list.
- The first method requires that the list is sorted according to natural ordering, whereas the second method requires that it is sorted according to the total ordering dictated by the comparator.
- The elements in the list and the key must also be mutually comparable.
- Successful searches return the index of the key in the list.
- A non-negative value indicates a successful search.
- Unsuccessful searches return a negative value given by the formula -(insertion point + 1), where insertion point is the index where the key would have been, had it been in the list.

Searching in Collections

- The following methods search for sublists:

```
int indexOfSubList(List<?> source, List<?> target)
int lastIndexOfSubList(List<?> source, List<?> target)
```

- These two methods find the first or last occurrence of the target list in the source list, respectively.
- They return the starting position of the target list in thesource list.
- The methods are applicable to lists of any type.

Searching in Collections

- The following methods find the minimum and maximum elements in a collection:

```
<E extends Object & Comparable<? super E>>
    E max(Collection<? extends E> c)
<E> E max(Collection<? extends E> c,
          Comparator<? super E> comp)
<E extends Object & Comparable<? super E>>
    E min(Collection<? extends E> c)
<E> E min(Collection<? extends E> cl,
          Comparator<? super E> comp)
```

Changing Elements in Collections

```
<E> boolean addAll(Collection<? super E> collection, E... elements)
```

- Adds the specified elements to the specified collection. Convenient method for loading a collection with a variable argument list or an array

```
<E> void copy(List<? super E> destination, List<? extends E> source)
```

- Adds the elements from the source list to the destination list.

```
<E> void fill(List<? super E> list, E element)
```

- Replaces all of the elements of the list with the specified element.

```
<E> boolean replaceAll(List<E> list, E oldVal, E newVal)
```

- Replaces all elements equal to oldVal with newVal in the list; returns true if the list was modified.

```
<E> List<E> nCopies(int n, E element)
```

- Creates an immutable list with n copies of the specified element.

Sorting Arrays

```
void sort(type[] array)
void sort(type[] array, int fromIndex, int toIndex)
<E> void sort(E[] array, Comparator<? super E> comp)
<E> void sort(E[] array, int fromIndex, int toIndex,
                    Comparator<? super E> comp)
```

Searching in Arrays

- The Arrays class provides enough overloaded versions of the binarySearch() method to search in practically any type of array that is sorted.
- The discussion on searching in lists is also applicable to searching in arrays.

Searching in Arrays

- The bounds, if specified in the methods below, define a half-open interval. The search is then confined to this interval.

```
int binarySearch(type[] array, type key)
int binarySearch(type[] array, int fromIndex, int toIndex, type key)
```

- Permitted type for elements include byte, char, double, float, int, long, short, and Object. In the case where an array of objects is passed as argument, the objects must be sorted in natural ordering, as defined by the Comparable interface.

```
<E> int binarySearch(E[] array, E key, Comparator<? super E> c)
<E> int binarySearch(E[] array, int fromIndex, int toIndex, E key,
                     Comparator<? super E> c)
```

- The two generic methods above require that the array is sorted according to the total ordering dictated by the comparator. In particular, its elements are mutually comparable according to this comparator. The comparator must be equivalent to the one that was used for sorting the array, otherwise the results are unpredictable.

Creating List Views of Arrays

- The asList() method in the Arrays class and the toArray() method in the Collection interface provide the bidirectional bridge between arrays and collections.
- The asList() method of the Arrays class creates List views of arrays.
- Changes to the List view reflect in the array, and vice versa. The List is said to be backed by the array. The List size is equal to the array length and cannot be changed.

<E> List<E> asList(E... elements)

- Returns a fixed-size list view backed by the array corresponding to the vararg argument elements.

Miscellaneous Utility Methods in the Arrays Class

```
void fill(type[] a,  type val)
void fill(type[] a, int fromIndex, int toIndex,
          type val)
```

- Assigns the specified value to each element of the specified array or specified range.

```
String toString(type [] a)
String deepToString(Object[] a)
```

- Returns a text representation of the contents (or “deep contents”) of the specified array

И это - все?

И это - все? Таки – ДА!