JAVA PROGRAMMING

Week 5:
The Collections Framework

Lecturers:

- HO Tuan Thanh, M.Sc.

Plan

1. Collections overview
2. Collection interfaces
3. Collection classes
4. Accessing a collection via an iterator
5. Spliterators
6. Storing user-defined classes in collections
7. Working with maps
8. Comparators

Plan

1. Collections overview
2. Collection interfaces
3. Collection classes
4. Accessing a collection via an iterator
5. Spliterators
6. Storing user-defined classes in collections
7. Working with maps
8. Comparators

java.util – Top-level classes [1]

AbstractCollection	FormattableFlags	Properties
AbstractList	Formatter	PropertyPermission
AbstractMap	GregorianCalendar	PropertyResourceBundle
AbstractQueue	HashMap	Random
AbstractSequentialList	HashSet	ResourceBundle
AbstractSet	Hashtable	Scanner
ArrayDeque	IdentityHashMap	ServiceLoader
ArrayList	IntSummaryStatistics	SimpleTimeZone
Arrays	LinkedHashMap	Spliterators
Base64	LinkedHashSet	SplitableRandom
BitSet	LinkedList	Stack

java.util – Top-level classes [2]

Calendar	ListResourceBundle	StringJoiner
Collections	Locale	StringTokenizer
Currency	LongSummaryStatistics	Timer
Date	Objects	TimerTask
Dictionary	Observable (Deprecated by JDK 9.)	TimeZone
DoubleSummaryStatistics	Optional	TreeMap
EnumMap	OptionalDouble	TreeSet
EnumSet	OptionalInt	UUID
EventListenerProxy	OptionalLong	Vector
EventObject	PriorityQueue	WeakHashMap

java.util – Interfaces

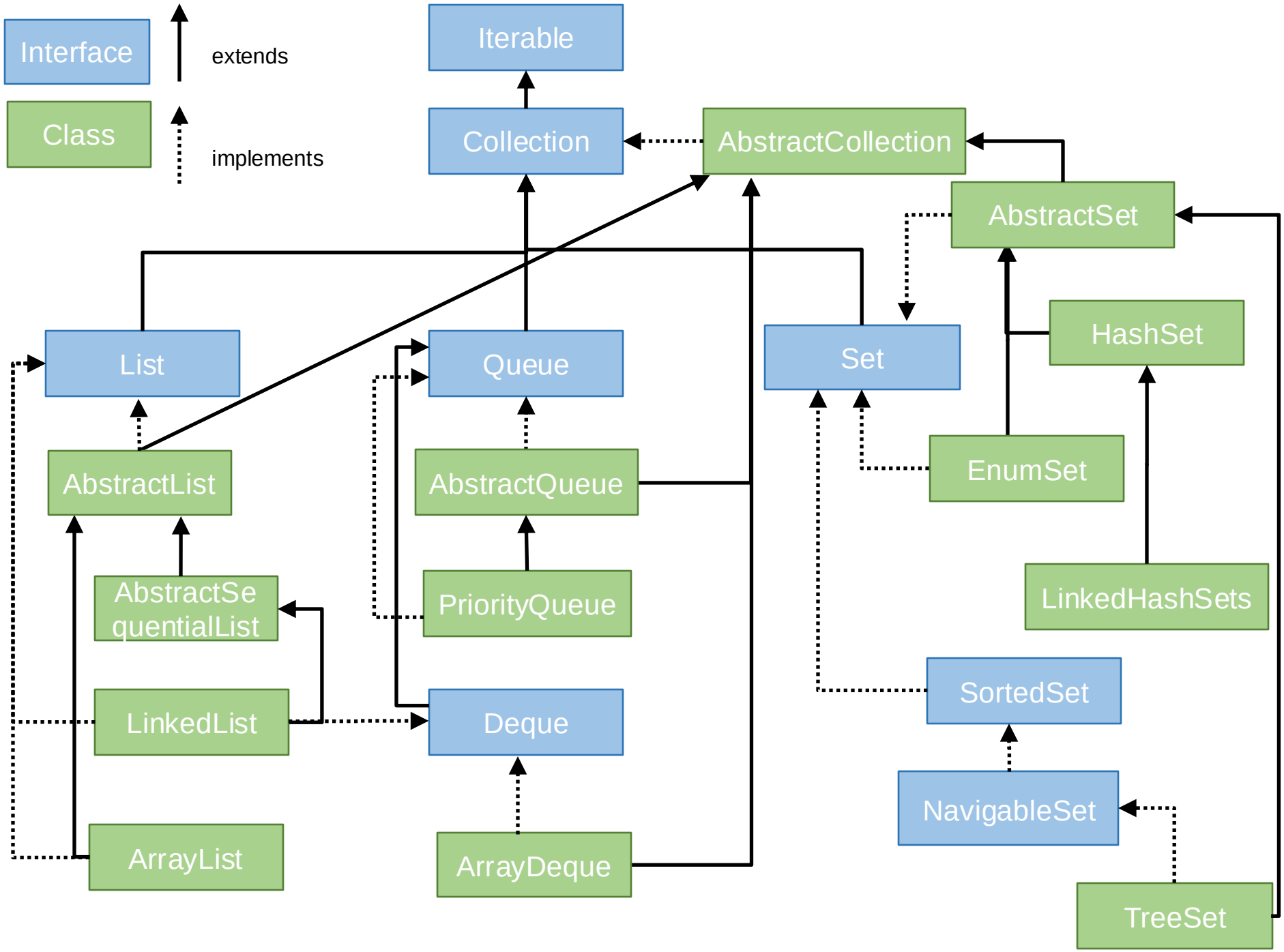
Collection	Map.Entry	ServiceLoader.Provider
Comparator	NavigableMap	Set
Deque	NavigableSet	SortedMap
Enumeration	Observer (Deprecated by JDK 9.)	SortedSet
EventListener	PrimitiveIterator	Spliterator
Formattable	PrimitiveIterator.OfDouble	Spliterator.OfDouble
Iterator	PrimitiveIterator.OfInt	Spliterator.OfInt
List	PrimitiveIterator.OfLong	Spliterator.OfLong
ListIterator	Queue	Spliterator.OfPrimitive
Map	RandomAccess	

Collections overview [1]

- The Java Collections Framework standardizes the way in which groups of objects are handled by your programs.
- The Collections Framework was designed to meet several goals.
  - High-performance.
  - Allow different types of collections to work in a similar manner and with a high degree of interoperability.
  - Easy to extend and/or adapt a collection.
  - Allow the integration of standard arrays into the Collections Framework.
- Algorithms are part of the collection mechanism.
- Provide the Iterator interface.

Collections overview [2]

- JDK 8 added another type of iterator called a **spliterator**.
- The framework also defines several map interfaces and classes.
  - Maps store key/value pairs.

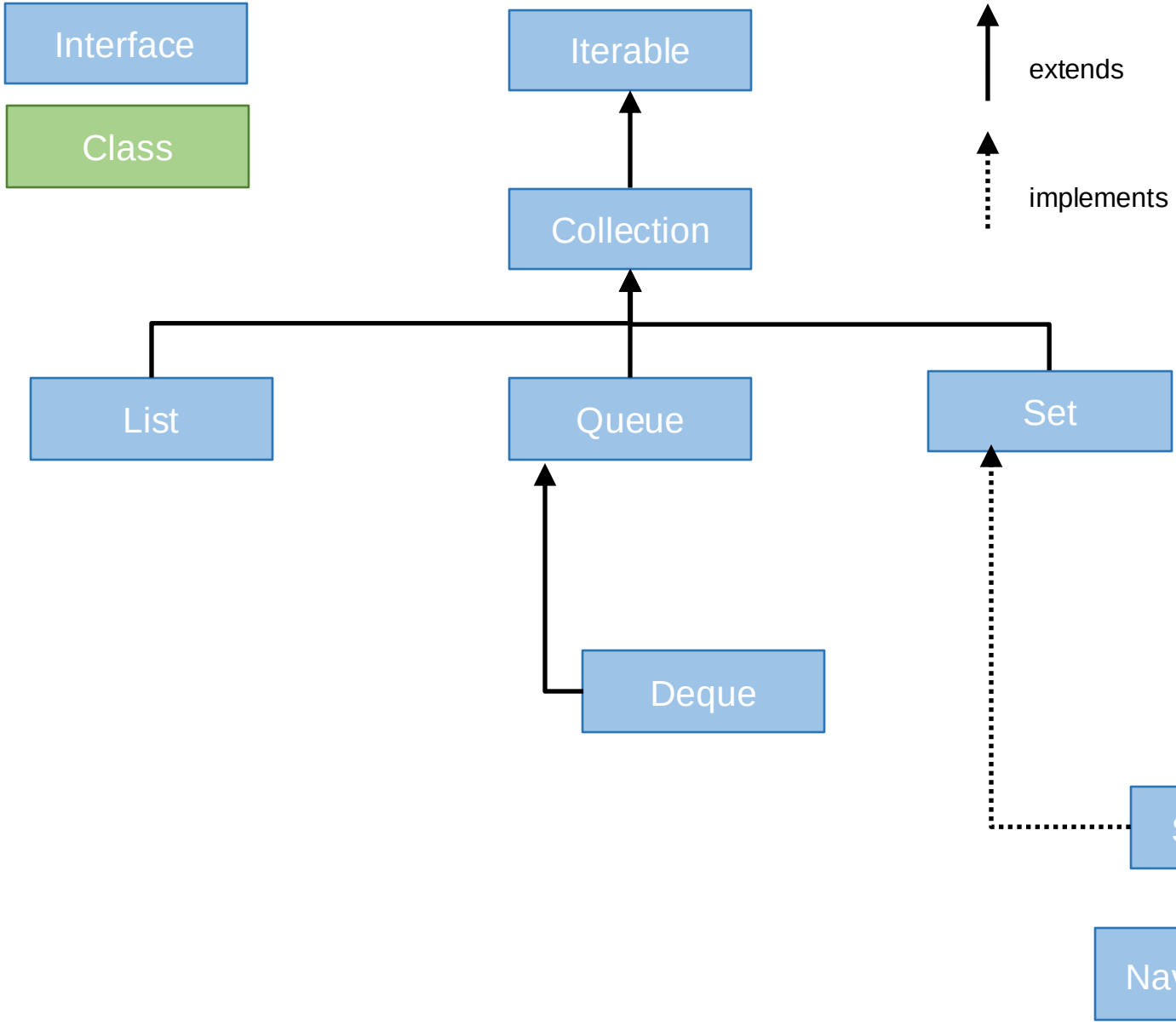

Interface
extends
Class
implements
Iterable
Collection
AbstractCollection
AbstractSet
HashSet
List
Queue
Set
AbstractList
AbstractQueue
EnumSet
AbstractSe
quentialList
PriorityQueue
LinkedHashSets
SortedSet
LinkedList
Deque
NavigableSet
ArrayList
ArrayDeque
TreeSet

Plan

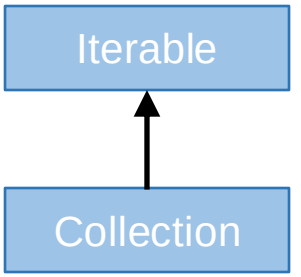
1. Collections overview
2. Collection interfaces
3. Collection classes
4. Accessing a collection via an iterator
5. Spliterators
6. Storing user-defined classes in collections
7. Working with maps
8. Comparators

The Collection Interfaces

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue.
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches.
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.


Interface
Class
Iterable
extends
implements
Collection
List
Queue
Set
Deque
SortedSet
NavigableSet

The Collection Interface



- The Collection interface is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection.

  interface Collection<E>

  - E specifies the type of objects that the collection will hold.
- Extends the Iterable interface.

  → All collections can be cycled through by using foreach.
- Methods declared by Collection interface can be found here: https://docs.oracle.com/javase/9/docs/api/java/util/Collection.html

Collection: Core methods [1]

Method	Description
public boolean add(E e)	It is used to insert an element in this collection.
public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
public boolean remove(Object element)	It is used to delete an element from the collection.
public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
default boolean removeIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
public int size()	It returns the total number of elements in the collection.
public void clear()	It removes the total number of elements from the collection.
public boolean contains(Object element)	It is used to search an element.

Collection: Core methods [2]

Method	Description
public Iterator iterator()	It returns an iterator.
public Object[] toArray()	It converts collection into array.
public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
public boolean isEmpty()	It checks if collection is empty.
default Stream<E> parallelStream()	It returns a possibly parallel Stream with the collection as its source.
default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.
public boolean equals(Object element)	It matches two collections.
public int hashCode()	It returns the hash code number of the collection.

The List Interface

- Declares the behavior of a collection that stores a sequence of elements.
  - Elements can be inserted or accessed by their position in the list, using a zero-based index.
  - A list may contain duplicate elements.
- Interface syntax:

```
interface List<E>
```

  - `E` specifies the type of objects that the list will hold.
- There are various methods in List interface that can be used to insert, delete, and access the elements from the list.
- For further information: https://docs.oracle.com/javase/9/docs/api/java/util/List.html

List: Core methods

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of a list.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
boolean equals(Object o)	It is used to compare the specified object with the elements of a list.
int hashcode()	It is used to return the hash code value for a list.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.

List: Core methods

Method	Description
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean contains(Object o)	It returns true if the list contains the specified element
boolean containsAll (Collection<?> c)	It returns true if the list contains all the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll (Collection<?> c)	It is used to remove all the elements from the list.

From JDK 9:
static <E> List<E> of()

- static <E> List<E> of()
- static <E> List<E> of(E e1)
- static <E> List<E> of(E... elements)
- static <E> List<E> of(E e1, E e2)
- static <E> List<E> of(E e1, E e2, E e3)
- ....
- static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)
- For all versions, null elements are not allowed. In all cases, the List implementation is unspecified.

The Set Interface

- Defines a set.
- Specifies the behavior of a collection that does not allow duplicate elements.
  - The add() method returns false if an attempt is made to add duplicate elements to a set.
  - With two exceptions, it does not specify any additional methods of its own.

interface Set<E>

  - E specifies the type of objects that the set will hold.

- From JDK 9: Set includes the of() factory method, which has a number of overloads.
- Each version returns an unmodifiable, value-based collection that is comprised of the arguments that it is passed.
- static <E> Set<E> of()
- static <E> Set<E> of(E e1)
- static <E> Set<E> of(E... elements)
- static <E> Set<E> of(E e1, E e2)
- ...
- static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)

- For all versions: null elements are not allowed. In all cases, the Set implementation is unspecified.
- From JDK 10:

```
static <E> Set<E> copyOf(Collection <? extends E> from)
```

  - Returns a set that contains the same elements as from.
  - Null values are not allowed.
  - The returned set is unmodifiable.

The SortedSet Interface

- Extends Set and declares the behavior of a set sorted in ascending order.

interface SortedSet<E>

  - E specifies the type of objects that the set will hold.

Methods declared by SortedSet

Methods	Description
Comparator<? super E> comparator()	Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.
E first()	Returns the first (lowest) element currently in this set.
SortedSet<E> headSet(E toElement)	Returns a view of the portion of this set whose elements are strictly less than toElement.
E last()	Returns the last (highest) element currently in this set.
default Spliterator<E> spliterator()	Creates a Spliterator over the elements in this sorted set.
SortedSet<E> subSet(E fromElement, E toElement)	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
SortedSet<E> tailSet(E fromElement)	Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

The NavigableSet Interface

- Extends SortedSet and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.

interface NavigableSet<E>

  - E specifies the type of objects that the set will hold.

Methods	Description
E ceiling(E e)	Returns the least element in this set greater than or equal to the given element, or null if there is no such element.
Iterator<E> descendingIterator()	Returns an iterator over the elements in this set, in descending order.
NavigableSet<E> descendingSet()	Returns a reverse order view of the elements contained in this set.
E floor(E e)	Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
SortedSet<E> headSet(E toElement)	Returns a view of the portion of this set whose elements are strictly less than toElement.
NavigableSet<E> headSet(E toElement, boolean inclusive)	Returns a view of the portion of this set whose elements are less than (or equal to, if inclusive is true) toElement.
E higher(E e)	Returns the least element in this set strictly greater than the given element, or null if there is no such element.
Iterator<E> iterator()	Returns an iterator over the elements in this set, in ascending order.
E lower(E e)	Returns the greatest element in this set strictly less than the given element, or null if there is no such element.

Methods	Description
E pollFirst()	Retrieves and removes the first (lowest) element, or returns null if this set is empty.
E pollLast()	Retrieves and removes the last (highest) element, or returns null if this set is empty.
NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	Returns a view of the portion of this set whose elements range from fromElement to toElement.
SortedSet<E> subSet(E fromElement, E toElement)	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
SortedSet<E> tailSet(E fromElement)	Returns a view of the portion of this set whose elements are greater than or equal to fromElement.
NavigableSet<E> tailSet(E fromElement, boolean inclusive)	Returns a view of the portion of this set whose elements are greater than (or equal to, if inclusive is true) fromElement.

The Queue Interface

- Extends Collection and declares the behavior of a queue, which is often a first-in, first-out list.
- However, there are types of queues in which the ordering is based upon other criteria.

interface Queue<E>

  - E specifies the type of objects that the queue will hold.

Methods declared by Queue

Methods	Description
boolean add(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.
E element()	Retrieves, but does not remove, the head of this queue.
boolean offer(E e)	Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
E peek()	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
E poll()	Retrieves and removes the head of this queue, or returns null if this queue is empty.
E remove()	Retrieves and removes the head of this queue.

The Deque Interface

- Deque is short for "double ended queue"
- Extends Queue and declares the behavior of a double ended queue.
- Can function as standard, first-in, first-out queues or as last-in, first-out stacks.

interface Deque<E>

  - E specifies the type of objects that the deque will hold.

Summary of Deque methods

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

Comparison of Queue and Deque methods

Queue Method	Equivalent Deque Method
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

Comparison of Stack and Deque methods

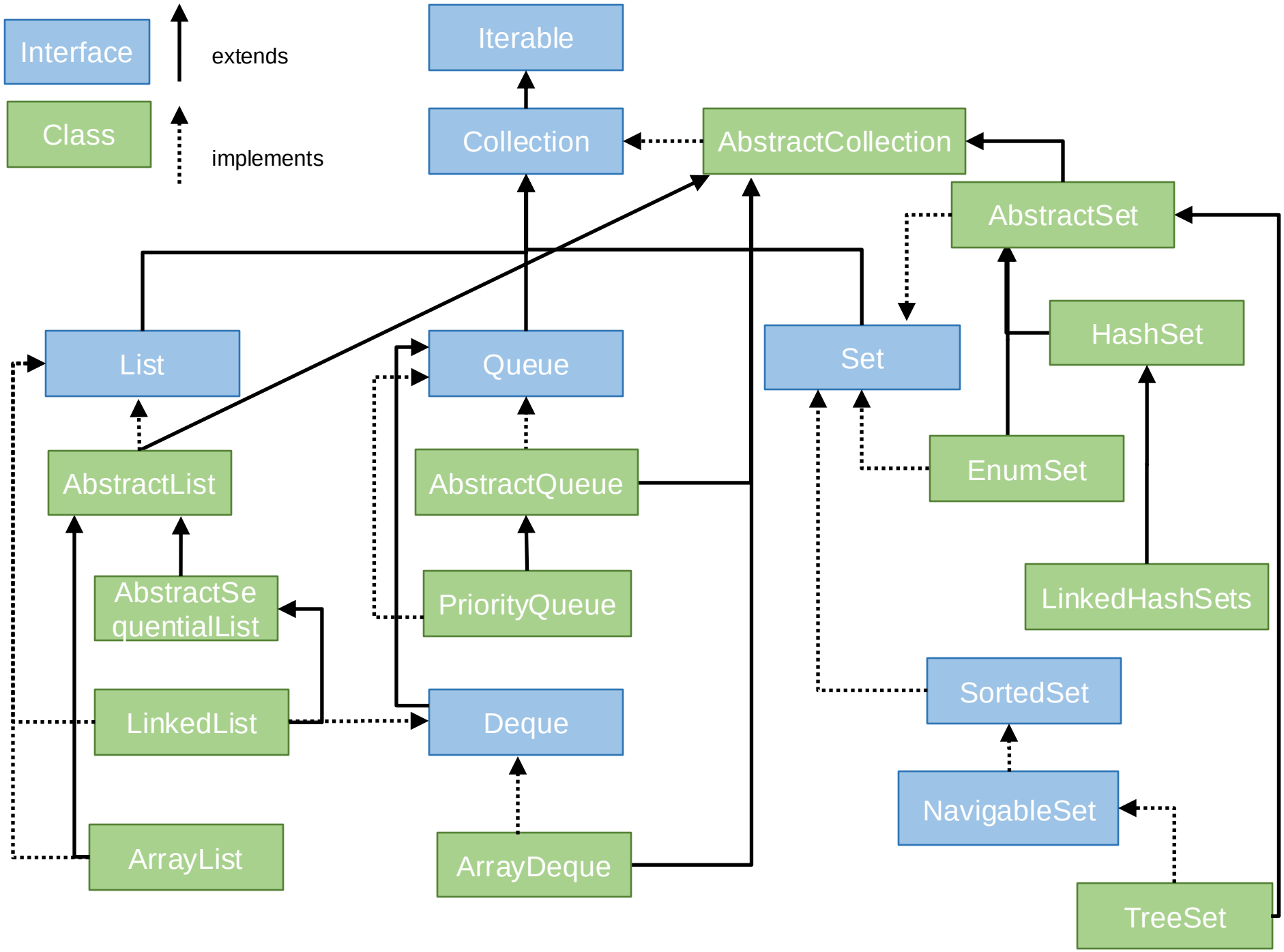
Stack Method	Equivalent Deque Method
push(e)	addFirst(e)
pop()	removeFirst()
peek()	peekFirst()

Plan

1. Collections overview
2. Collection interfaces
3. Collection classes
4. Accessing a collection via an iterator
5. Spliterators
6. Storing user-defined classes in collections
7. Working with maps
8. Comparators

The Collection Classes

Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface.
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.


Interface
extends
Class
implements
Iterable
Collection
AbstractCollection
AbstractSet
HashSet
List
Queue
Set
AbstractList
AbstractQueue
EnumSet
AbstractSe
quentialList
PriorityQueue
LinkedHashSets
SortedSet
LinkedList
Deque
NavigableSet
ArrayList
ArrayDeque
TreeSet

The ArrayList Class

- Extends AbstractList and implements the List interface.

  class ArrayList<E>

  - E specifies the type of objects that the list will hold.
- Supports dynamic arrays that can grow as needed.
- Is a variable-length array of object references
  - Array lists are created with an initial size.
  - When this size is exceeded, the collection is automatically enlarged.

Constructors

- Constructors:

  ArrayList()
  ArrayList(Collection<? extends E> c)
  ArrayList(int capacity)

```
1.  import java.util.*;
2.  class ArrayListDemo {
3.      public static void main(String args[]) {
4.      ArrayList<String> al = new ArrayList<String>();
5.      System.out.println("Initial size of al: " + al.size());
6.      al.add("C"); al.add("A"); al.add("E"); al.add("B");
7.      al.add("D"); al.add("F"); al.add(1, "A2");
8.      System.out.println("Size of al after additions: " +
9.                                                      al.size());
10.     System.out.println("Contents of al: " + al);
11.
12.     al.remove("F"); al.remove(2);
13.     System.out.println("Size of al after deletions: " +
14.                                                     al.size());
15.     System.out.println("Contents of al: " + al);
16.     }
17. }
```

Obtaining an Array from an ArrayList

object[ ] toArray( )

<T> T[ ] toArray(T array[ ])

default <T> T[] toArray(IntFunction<T[]> arrayGen)

```
1.  //Convert an ArrayList into an array.
2.  public class ArrayListToArray {
3.      public static void main(String args[]) {
4.          // Create an array list.
5.          ArrayList<Integer> al = new ArrayList<Integer>();
6.          // Add elements to the array list.
7.          al.add(1); al.add(2); al.add(3); al.add(4);
8.          System.out.println("Contents of al: " + al);
9.          // Get the array.
10.         Integer ia[] = new Integer[al.size()];
11.         ia = al.toArray(ia);
12.         int sum = 0;
13.         // Sum the array.
14.         for (int i : ia) sum += i;
15.         System.out.println("Sum is: " + sum);
16.     }
17. }
```

The LinkedList Class

- Extends AbstractSequentialList and implements the List, Deque, and Queue interfaces. It provides a linked-list data structure.

  class LinkedList<E>

  - E specifies the type of objects that the list will hold.
- Constructors :

  LinkedList( )

  LinkedList(Collection<? extends E> c)

```
1.  //Demonstrate LinkedList.
2.  import java.util.*;
3.  class LinkedListDemo {
4.      public static void main(String args[]) {
5.          // Create a linked list.
6.          LinkedList<String> ll = new LinkedList<String>();
7.          // Add elements to the linked list.
8.          ll.add("F"); ll.add("B"); ll.add("D"); ll.add("E");
9.          ll.add("C"); ll.addLast("Z"); ll.addFirst("A");
10.         ll.add(1, "A2");
11.         System.out.println("Original contents of ll: " + ll);
12.         // Remove elements from the linked list.
```

```
1.              ll.remove("F"); ll.remove(2);
2.              System.out.println("Contents of ll after deletion: " +
3.                                                                  ll);
4.              // Remove first and last elements.
5.              ll.removeFirst(); ll.removeLast();
6.              System.out.println("ll after deleting first and last:”
7.                                                  + ll);
8.              // Get and set a value.
9.              String val = ll.get(2);
10.             ll.set(2, val + " Changed");
11.             System.out.println("ll after change: " + ll);
12.         }
13.     }
```

The HashSet Class

- Extends AbstractSet and implements the Set interface.
- Creates a collection that uses a hash table for storage.

class HashSet<E>

  - E specifies the type of objects that the set will hold.

- Constructors:

HashSet( )

HashSet(Collection<? extends E>c) HashSet(int capacity)

HashSet(int capacity, float fillRatio)

Example


```
1.  //Demonstrate HashSet.
2.  import java.util.*;
3.  class HashSetDemo {
4.      public static void main(String args[]) {
5.          // Create a hash set.
6.          HashSet<String> hs = new HashSet<String>();
7.          // Add elements to the hash set.
8.          hs.add("Beta"); hs.add("Alpha");
9.          hs.add("Eta"); hs.add("Gamma");
10.         hs.add("Epsilon"); hs.add("Omega");
11.         hs.add("Eta");
12.
13.         System.out.println(hs);
14.     }
    }
```

The LinkedHashSet Class

- Extends HashSet and adds no members of its own.

  class LinkedHashSet<E>

  - E specifies the type of objects that the set will hold.
- Its constructors parallel those in HashSet.
- Maintains a linked list of the entries in the set, in the order in which they were inserted.

The TreeSet Class

- Extends AbstractSet and implements the NavigableSet interface.
- Creates a collection that uses a tree for storage; Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast → excellent choice when storing large amounts of sorted information that must be found quickly.

class TreeSet<E>

  - E specifies the type of objects that the set will hold.

- Constructors:
  - TreeSet( )
  - TreeSet(Collection<? extends E> c)
  - TreeSet(Comparator<? super E> comp)
  - TreeSet(SortedSet<E> ss)

Example


```
1.  //Demonstrate TreeSet.
2.  import java.util.*;
3.  class TreeSetDemo {
4.      public static void main(String args[]) {
5.          // Create a tree set.
6.          TreeSet<String> ts = new TreeSet<String>();
7.          // Add elements to the tree set.
8.          ts.add("C"); ts.add("A"); ts.add("B");
9.          ts.add("E"); ts.add("F"); ts.add("D");
10.
11.         System.out.println(ts);
12.     }
13. }
```

The PriorityQueue Class

- Extends AbstractQueue and implements the Queue interface.
- Creates a queue that is prioritized based on the queue's comparator.

```
class PriorityQueue<E>
```

  - E specifies the type of objects stored in the queue.
- PriorityQueues are dynamic, growing as necessary.

- Constructors:

PriorityQueue( )

PriorityQueue(int capacity)

PriorityQueue(Comparator<? super E> comp)

PriorityQueue(int capacity, Comparator<? super E> comp)

PriorityQueue(Collection<? extends E> c)

PriorityQueue(PriorityQueue<? extends E> c)

PriorityQueue(SortedSet<? extends E> c)

The ArrayDeque Class

- Extends AbstractCollection and implements the Deque interface.
- It adds no methods of its own.
- ArrayDeque creates a dynamic array and has no capacity restrictions.

class ArrayDeque<E>

  - E specifies the type of objects stored in the collection.

- Constructors:

  ArrayDeque()

  ArrayDeque(int size)

  ArrayDeque(Collection<? extends E> c)

```
1.  import java.util.*;
2.  class ArrayDequeDemo {
3.      public static void main(String args[]) {
4.          // Create a tree set.
5.          ArrayDeque<String> adq = new ArrayDeque<String>();
6.          // Use an ArrayDeque like a stack.
7.          adq.push("A"); adq.push("B"); adq.push("D");
8.          adq.push("E"); adq.push("F");
9.          System.out.print("Popping the stack: ");
10.         while (adq.peek() != null)
11.             System.out.print(adq.pop() + " ");
12.         System.out.println();
13.     }
14. }
```

The EnumSet Class

- Extends AbstractSet and implements Set.
- It is specifically for use with elements of an enum type.

```
class EnumSet<E extends Enum<E>>
```

  - E specifies the elements.
  - Notice that E must extend Enum<E>, which enforces the requirement that the elements must be of the specified enum type.
- EnumSet defines no constructors.

The EnumSet methods [1]

Methods	Description
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)	Creates an enum set containing all of the elements in the specified element type.
EnumSet<E> clone()	Returns a copy of this set.
static <E extends Enum<E>>EnumSet<E> complementOf(EnumSet<E> s)	Creates an enum set with the same element type as the specified enum set, initially containing all the elements of this type that are *not* contained in the specified set.
static <E extends Enum<E>> EnumSet<E>copyOf(Collection<E>c)	Creates an enum set initialized from the specified collection.
static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)	Creates an enum set with the same element type as the specified enum set, initially containing the same elements (if any).
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType)	Creates an empty enum set with the specified element type.

The EnumSet methods [2]

Methods	Description
static <E extends Enum<E>>EnumSet<E> of(E e)	Creates an enum set initially containing the specified element.
static <E extends Enum<E>> EnumSet<E> of(E e1, E e2)	Creates an enum set initially containing the specified elements.
static <E extends Enum<E>> EnumSet<E> of(E first, E... rest)	Creates an enum set initially containing the specified elements.
....	.....
static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)	Creates an enum set initially containing the specified elements.
static <E extends Enum<E>> EnumSet<E> range(E from, E to)	Creates an enum set initially containing all of the elements in the range defined by the two specified endpoints.

Plan

1. Collections overview
2. Collection interfaces
3. Collection classes
4. Accessing a collection via an iterator
5. Spliterators
6. Storing user-defined classes in collections
7. Working with maps
8. Comparators

Accessing a Collection via an Iterator

- To cycle through the elements in a collection: employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.
  - Iterator enables you to cycle through a collection, obtaining or removing elements.
  - ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

  interface Iterator<E>

  interface ListIterator<E>
  - E specifies the type of objects being iterated.

Itertor methods

Methods	Description
default void forEachRemaining(Consumer<? super E> action)	Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
boolean hasNext()	Returns true if the iteration has more elements.
E next()	Returns the next element in the iteration.
default void remove()	Removes from the underlying collection the last element returned by this iterator (optional operation).

Methods	Description
void add(E e)	Inserts the specified element into the list.
boolean hasNext()	Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean hasPrevious()	Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E next()	Returns the next element in the list and advances the cursor position.
int nextIndex()	Returns the index of the element that would be returned by a subsequent call to next().
E previous()	Returns the previous element in the list and moves the cursor position backwards.
int previousIndex()	Returns the index of the element that would be returned by a subsequent call to previous().
void remove()	Removes from the list the last element that was returned by next() or previous() (optional operation).
void set(E e)	Replaces the last element returned by next() or previous() with the specified element.

Using an Iterator

1. Obtain an iterator to the start of the collection by calling the collection's iterator() method.
2. Set up a loop that makes a call to hasNext(). Have the loop iterate as long as hasNext() returns true.
3. Within the loop, obtain each element by calling next().

```
1.  class IteratorDemo {
2.      public static void main(String args[]) {
3.          ArrayList<String> al = new ArrayList<String>();
4.          al.add("C"); al.add("A"); al.add("E"); al.add("B");
5.          al.add("D"); al.add("F");
6.          System.out.print("Original contents of al: ");
7.          Iterator<String> itr = al.iterator();
8.          while (itr.hasNext()) {
9.              String element = itr.next();
10.             System.out.print(element + " ");
11.         }
12.         System.out.println();
```

```
1.              ListIterator<String> litr = al.listIterator();
2.              while (litr.hasNext()) {
3.                      String element = litr.next();
4.                      litr.set(element + "+");
5.              }
6.              System.out.print("Modified contents of al: ");
7.              itr = al.iterator();
8.              while (itr.hasNext()) {
9.                      String element = itr.next();
10.                     System.out.print(element + " ");
11.             }  System.out.println();
12.             System.out.print("Modified list backwards: ");
13.             while (litr.hasPrevious()) {
14.                     String element = litr.previous();
15.                     System.out.print(element + " ");
16.             }  System.out.println();
17.     }
18. }
```

Using For-Each


```
1.  class ForEachDemo {
2.      public static void main(String args[]) {
3.          ArrayList<Integer> vals = new ArrayList<Integer>();
4.          vals.add(1); vals.add(2); vals.add(3);
5.          vals.add(4); vals.add(5);
6.          System.out.print("Original contents of vals: ");
7.          for (int v : vals) System.out.print(v + " ");
8.          System.out.println();
9.          // Now, sum the values by using a for loop.
10.         int sum = 0;
11.         for (int v : vals) sum += v;
12.         System.out.println("Sum of values: " + sum);
        }
13. }
```

Plan

1. Collections overview
2. Collection interfaces
3. Collection classes
4. Accessing a collection via an iterator
5. Spliterators
6. Storing user-defined classes in collections
7. Working with maps
8. Comparators

Spliterators

- JDK 8 added another type of iterator: **spliterator** – defined by the Spliterator interface.
- A spliterator cycles through a sequence of elements, similar to the iterators.
- Spliterator supports parallel programming.
  - Use Spliterator even if you won't be using parallel execution.
  - One reason: it offers a streamlined approach that combines the hasNext and next operations into one method.

interface Spliterator<T>

  - T is the type of elements being iterated.

int characteristics()	Returns a set of characteristics of this Spliterator and its elements.
long estimateSize()	Returns an estimate of the number of elements that would be encountered by a forEachRemaining, or returns Long.MAX_VALUE if infinite, unknown, or too expensive to compute.
default void forEachRemaining(Consumer<? super T> action)	Performs the given action for each remaining element, sequentially in the current thread, until all elements have been processed or the action throws an exception.
default Comparator<? super T> getComparator()	If this Spliterator's source is SORTED by a Comparator, returns that Comparator.
default long getExactSizeIfKnown()	Convenience method that returns estimateSize() if this Spliterator is SIZED, else -1.
default boolean hasCharacteristics(int characteristics)	Returns true if this Spliterator's characteristics() contain all of the given characteristics.
boolean tryAdvance(Consumer<? super T> action)	If a remaining element exists, performs the given action on it, returning true; else returns false.
Spliterator<T> trySplit()	If this spliterator can be partitioned, returns a Spliterator covering elements, that will, upon return from this method, not be covered by this Spliterator.

```
1.  import java.util.*;
2.  class SpliteratorDemo {
3.      public static void main(String args[]) {
4.          // Create an array list for doubles.
5.          ArrayList<Double> vals = new ArrayList<>();
6.          // Add values to the array list.
7.          vals.add(1.0); vals.add(2.0); vals.add(3.0);
8.          vals.add(4.0); vals.add(5.0);
9.          // Use tryAdvance() to display contents of vals.
10.         System.out.print("Contents of vals:\n");
11.         Spliterator<Double> spltitr = vals.spliterator();
12.         while (spltitr.tryAdvance((n) ->        System.out.println(n)));
13.
```

```
1.              System.out.println();
2.              // Create new list that contains square roots.
3.              spltitr = vals.spliterator();
4.              ArrayList<Double> sqrs = new ArrayList<>();
5.              while (spltitr.tryAdvance((n) ->
6.                                              sqrs.add(Math.sqrt(n))));
7.              // Use forEachRemaining to display contents of sqrs.
8.              System.out.print("Contents of sqrs:\n");
9.              spltitr = sqrs.spliterator();
10.             spltitr.forEachRemaining((n) ->
11.                                             System.out.println(n));
12.             System.out.println();
13.         }
14.     }
```

Plan

1. Collections overview
2. Collection interfaces
3. Collection classes
4. Accessing a collection via an iterator
5. Spliterators
6. Storing user-defined classes in collections
7. Working with maps
8. Comparators

Storing User-Defined Classes in Collections [1]


```
1.  class Address {
2.      private String name; private String street;
3.      private String city; private String state;
4.      private String code;
5.      Address(String n, String s, String c,
6.                                          String st, String cd) {
7.          name = n; street = s; city = c;
8.          state = st; code = cd;
9.          }
10.         public String toString() {
11.         return name + "\n" + street + "\n" + city + " " +
12.                                          state + " " + code;
13.     }
14. }
```

Storing User-Defined Classes in Collections [2]


```
1.  class MailList {
2.      public static void main(String args[]) {
3.          LinkedList<Address> ml = new LinkedList<Address>();
4.          ml.add(new Address("J.W. West", "11 Oak Ave",
5.                                          "Urbana", "IL", "61801"));
6.          ml.add(new Address("Ralph Baker", "1142 Maple    Lane",
7.                                          "Mahome", "IL", "61853"));
8.          ml.add(new Address("Tom Carlton", "867 Elm St",
9.                                          "Champaign", "IL", "61820"));
10.         for (Address element : ml)
11.             System.out.println(element + "\n");
12.         System.out.println();
13.     }
14. }
```

Plan

1. Collections overview
2. Collection interfaces
3. Collection classes
4. Accessing a collection via an iterator
5. Spliterators
6. Storing user-defined classes in collections
7. Working with maps
8. Comparators

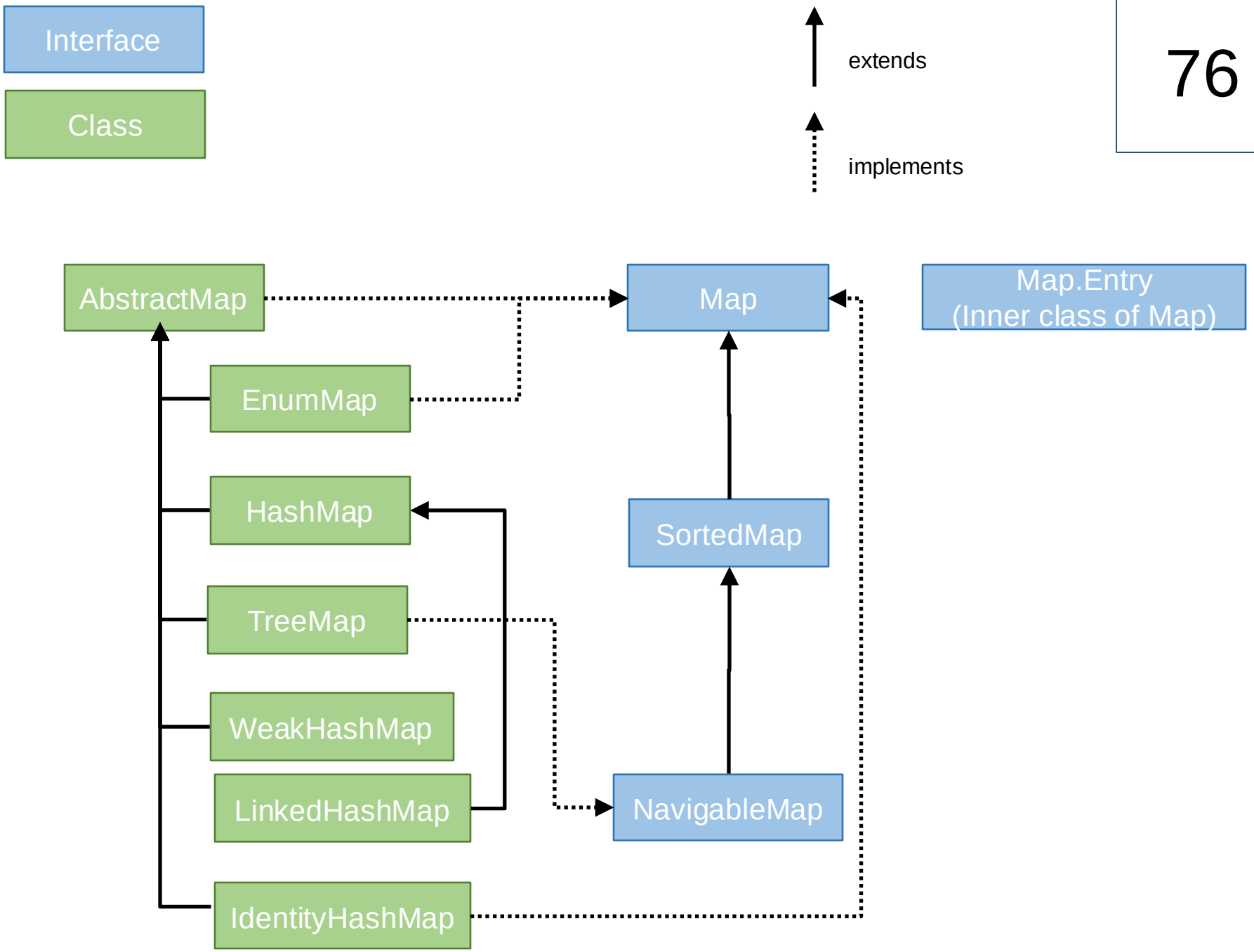
Working with Maps

- A map is an object that stores associations between keys and values, or key/value pairs.
  - Given a key, you can find its value.
  - Both keys and values are objects.
  - The keys must be unique, but the values may be duplicated.
  - Some maps can accept a null key and null values, others cannot.
- Maps don't implement the Iterable interface.
  - You cannot cycle through a map using a for-each style for loop.
  - You can't obtain an iterator to a map.

The Map Interfaces

- Interfaces support maps:

Interface	Description
Map	Maps unique keys to values
Map.Entry	Describes an element (key-value pair) in a map. This is an inner class of Map.
NavigableMap	Extends SortedMap to handle the retrieval of entries based on closest-match searches
SortedMap	Extends Map so that the keys are maintained in ascending order


Interface
Class
extends
implements
AbstractMap
Map
Map.Entry
(Inner class of Map)
EnumMap
HashMap
SortedMap
TreeMap
WeakHashMap
LinkedHashMap
NavigableMap
IdentityHashMap

The Map Interface

- The Map interface maps unique keys to values.
  - A key is an object that you use to retrieve a value.
  - Given a key and a value, you can store the value in a Map object.
  - After the value is stored, you can retrieve it by using its key.

interface Map<K, V>

  - **K** specifies the type of keys, and
  - **V** specifies the type of values.
- Methods defined in Map can be found here:

https://docs.oracle.com/javase/9/docs/api/java/util/Map.html

The SortedMap Interface

- Extends Map.
- It ensures that the entries are maintained in ascending order based on the keys.

interface SortedMap<K, V>

  - K specifies the type of keys, and
  - V specifies the type of values.

Methods declared by SortedMap

Method	Description
Comparator<? super K> comparator( )	Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, **null** is returned.
K firstKey( )	Returns the first key in the invoking map.
SortedMap<K, V> headMap(K *end*)	Returns a sorted map for those map entries with keys that are less than *end*.
K lastKey( )	Returns the last key in the invoking map.
SortedMap<K, V> subMap(K *start*, K *end*)	Returns a map containing those entries with keys that are greater than or equal to *start* and less than *end*.
SortedMap<K, V> tailMap(K *start*)	Returns a map containing those entries with keys that are greater than or equal to *start*.

Methods	Description
Comparator<? super K> comparator()	Returns the comparator used to order the keys in this map, or null if this map uses the natural ordering of its keys.
Set<Map.Entry<K,V>> entrySet()	Returns a Set view of the mappings contained in this map.
K firstKey()	Returns the first (lowest) key currently in this map.
SortedMap<K,V> headMap(K toKey)	Returns a view of the portion of this map whose keys are strictly less than toKey.
Set<K> keySet()	Returns a Set view of the keys contained in this map.
K lastKey()	Returns the last (highest) key currently in this map.
SortedMap<K,V> subMap(K fromKey, K toKey)	Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
SortedMap<K,V> tailMap(K fromKey)	Returns a view of the portion of this map whose keys are greater than or equal to fromKey.
Collection<V> values()	Returns a Collection view of the values contained in this map.

The NavigableMap Interface

- Extends SortedMap and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys.

interface NavigableMap<K,V>

  - **K** specifies the type of the keys, and
  - **V** specifies the type of the values associated with the keys.

- Methods defined by NavigableMaps can be found here:

https://docs.oracle.com/javase/9/docs/api/java/util/NavigableMap.html

The Map.Entry Interface

- Work with a map entry.
  - Example: recall that the entrySet() method declared by the Map interface returns a Set containing the map entries.
  - Each of these set elements is a Map.Entry object.

interface Map.Entry<K, V>

  - K specifies the type of keys, and V specifies the type of values.

Method	Description
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map.Entry** whose key and value are equal to that of the invoking object.
K getKey( )	Returns the key for this map entry.
V getValue( )	Returns the value for this map entry.
int hashCode( )	Returns the hash code for this map entry.
V setValue(V *v*)	Sets the value for this map entry to *v*. A **ClassCastException** is thrown if *v* is not the correct type for the map. An **IllegalArgumentException** is thrown if there is a problem with *v*. A **NullPointerException** is thrown if *v* is **null** and the map does not permit **null** keys. An **UnsupportedOperationException** is thrown if the map cannot be changed.

The Map Classes

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.

The HashMap Class

- Extends AbstractMap and implements the Map interface.
  - It uses a hash table to store the map.
  - This allows the execution time of get() and put() to remain constant even for large sets.

    class HashMap<K, V>
  - K specifies the type of keys, and V specifies the type of values.
- Constructors:
  - HashMap()
  - HashMap(Map<? extends K, ? extends V> m)
  - HashMap(int capacity)
  - HashMap(int capacity, float fillRatio)

Example [1]


```
1.  import java.util.*;
2.  class HashMapDemo {
3.      public static void main(String args[]) {
4.          HashMap<String, Double> hm = new HashMap<String,
5.                                                   Double>();
6.          hm.put("John Doe", new Double(3434.34));
7.          hm.put("Tom Smith", new Double(123.22));
8.          hm.put("Jane Baker", new Double(1378.00));
9.          hm.put("Tod Hall", new Double(99.22));
10.         hm.put("Ralph Smith", new Double(-19.08));
11.
12.
```

Example [2]


```
1.              Set<Map.Entry<String, Double>> set = hm.entrySet();
2.              for (Map.Entry<String, Double> me : set) {
3.                  System.out.print(me.getKey() + ": ");
4.                  System.out.println(me.getValue());
5.              }
6.              System.out.println();
7.              double balance = hm.get("John Doe");
8.              hm.put("John Doe", balance + 1000);
9.              System.out.println("John Doe's new balance: " +
10.              hm.get("John Doe"));
11.         }
12. }
```

The TreeMap Class [1]

- Extends AbstractMap and implements the NavigableMap interface.
- It creates maps stored in a tree structure.
- A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval.
  - Unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order. T

```
class TreeMap<K, V>
```

  - K specifies the type of keys, and
  - V specifies the type of values.

The TreeMap Class [2]

- Constructors:
  - TreeMap()
  - TreeMap(Comparator<? super K> comp)
  - TreeMap(Map<? extends K, ? extends V> m)
  - TreeMap(SortedMap<K, ? extends V> sm)

Example


```
import java.util.*;
class TreeMapDemo {
    public static void main(String args[]) {
        TreeMap<String, Double> tm = new TreeMap<String,
                                                 Double>();
        // Put elements to the map.
        tm.put("John Doe", new Double(3434.34));
        tm.put("Tom Smith", new Double(123.22));
        tm.put("Jane Baker", new Double(1378.00));
        tm.put("Tod Hall", new Double(99.22));
        tm.put("Ralph Smith", new Double(-19.08));
```

Example


```
1.              Set<Map.Entry<String, Double>> set = tm.entrySet();
2.              for (Map.Entry<String, Double> me : set) {
3.                  System.out.print(me.getKey() + ": ");
4.                  System.out.println(me.getValue());
5.              }
6.              System.out.println();
7.              double balance = tm.get("John Doe");
8.              tm.put("John Doe", balance + 1000);
9.              System.out.println("John Doe's new balance: " +
10.                                                  tm.get("John Doe"));
11.         }
12. }
```

The LinkedHashMap Class

- Extends HashMap.
- It maintains a linked list of the entries in the map, in the order in which they were inserted.
  - This allows insertion-order iteration over the map.
  - When iterating through a collectionview of a LinkedHashMap, the elements will be returned in the order in which they were inserted.
  - You can also create a LinkedHashMap that returns its elements in the order in which they were last accessed.

class LinkedHashMap<K, V>

  - K specifies the type of keys, and
  - V specifies the type of values.

- Constructors:
  - LinkedHashMap( )
  - LinkedHashMap(Map<? extends K, ? extends V> m)
  - LinkedHashMap(int capacity)
  - LinkedHashMap(int capacity, float fillRatio)
  - LinkedHashMap(int capacity, float fillRatio, boolean Order)
- LinkedHashMap adds only one method to those defined by HashMap:

protected boolean removeEldestEntry(Map.Entry<K, V> e)

  - This method is called by put() and putAll().
  - The oldest entry is passed in **e**. By default, this method returns false and does nothing.

The IdentityHashMap Class

- Extends AbstractMap and implements the Map interface.
- Similar to HashMap except that it uses reference equality when comparing elements.
- Is a generic class that has this declaration:

```
class IdentityHashMap<K, V>
```

  - K specifies the type of key, and
  - V specifies the type of value.
- The API documentation explicitly states that IdentityHashMap is not for general use.

The EnumMap Class

- Extends AbstractMap and implements Map.
- Specifically for use with keys of an enum type.
- Declaration:

  class EnumMap<K extends Enum<K>, V>

  - K specifies the type of key, and V specifies the type of value.
  - K must extend Enum<K>, which enforces the requirement that the keys must be of an enum type.
- Constructors:

EnumMap(Class<K> kType) EnumMap(Map<K, ? extends V> m)

EnumMap(EnumMap<K, ? extends V> em)

- Defines no methods of its own.

Plan

1. Collections overview
2. Collection interfaces
3. Collection classes
4. Accessing a collection via an iterator
5. Spliterators
6. Storing user-defined classes in collections
7. Working with maps
8. Comparators

Comparators [1]

- Both TreeSet and TreeMap store elements in sorted order.
- It is the comparator that defines precisely what "sorted order" means.
- By default, these classes store their elements by using what Java refers to as "natural ordering"
- To order elements a different way: then specify a Comparator when you construct the set or map → govern precisely how elements are stored within sorted collections and maps.

interface Comparator<T>

  - T specifies the type of objects being compared.

Comparators [2]

- Prior to JDK 8: the Comparator interface defined only two methods:

```
int compare(T obj1, T obj2)
```

  - obj1 and obj2 are the objects to be compared.

```
boolean equals(object obj)
```

  - obj is the object to be tested for equality.
- JDK 8 added significant new functionality to Comparator through the use of default and static interface methods.

Methods	Description
int compare(T o1, T o2)	Compares its two arguments for order.
boolean equals(Object obj)	Indicates whether some other object is "equal to" this comparator.
static <T extends Comparable<? super T>>Comparator<T> naturalOrder()	Returns a comparator that compares Comparable objects in natural order.
static <T> Comparator<T> nullsFirst (Comparator<? super T> comparator)	Returns a null-friendly comparator that considers null to be less than non-null.
static <T> Comparator<T> nullsLast (Comparator<? super T> comparator)	Returns a null-friendly comparator that considers null to be greater than non-null.
default Comparator<T> reversed()	Returns a comparator that imposes the reverse ordering of this comparator.
static <T extends Comparable<? super T>>Comparator<T> reverseOrder()	Returns a comparator that imposes the reverse of the natural ordering.

Methods	Description
static <T,U extends Comparable<? super U>>Comparator<T> **comparing**(Function<? super T,? extends U> keyExtractor)	Accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T,U> Comparator<T> **comparing**(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)	Accepts a function that extracts a sort key from a type T, and returns a Comparator<T> that compares by that sort key using the specified Comparator.
static <T> Comparator<T> **comparingDouble**(ToDoubleFunction<? super T> keyExtractor)	Accepts a function that extracts a double sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T> Comparator<T> **comparingInt**(ToIntFunction<? super T> keyExtractor)	Accepts a function that extracts an int sort key from a type T, and returns a Comparator<T> that compares by that sort key.
static <T> Comparator<T> **comparingLong**(ToLongFunction<? super T> keyExtractor)	Accepts a function that extracts a long sort key from a type T, and returns a Comparator<T> that compares by that sort key.

Methods	Description
default Comparator<T> thenComparing(Comparator<? super T> other)	Returns a lexicographic-order comparator with another comparator.
default <U extends Comparable<? super U>>Comparator<T>thenComparing (Function<? super T,? extends U> keyExtractor)	Returns a lexicographic-order comparator with a function that extracts a Comparable sort key.
default <U> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)	Returns a lexicographic-order comparator with a function that extracts a key to be compared with the given Comparator.
default Comparator<T> thenComparingDouble(ToDoubleFunction<? super T> keyExtractor)	Returns a lexicographic-order comparator with a function that extracts a double sort key.
default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor)	Returns a lexicographic-order comparator with a function that extracts an int sort key.
default Comparator<T> thenComparingLong(ToLongFunction<? super T> keyExtractor)	Returns a lexicographic-order comparator with a function that extracts a long sort key.

Example: Using a Comparator [1]


```
1.  //Use a custom comparator.
2.  import java.util.*;
3.  //A reverse comparator for strings.
4.  class MyComp implements Comparator<String> {
5.      public int compare(String aStr, String bStr) {
6.          // Reverse the comparison.
7.          return bStr.compareTo(aStr);
8.      }
9.      // No need to override equals or the default methods.
10. }
```

Example: Using a Comparator [2]


```
1.  import java.util.TreeSet;
2.  class CompDemo {
3.      public static void main(String args[]) {
4.          TreeSet<String> ts = new
5.                                  TreeSet<String>(new MyComp());
6.          ts.add("C"); ts.add("A"); ts.add("B");
7.          ts.add("E"); ts.add("F"); ts.add("D");
8.          // Display the elements.
9.          for (String element : ts)
10.             System.out.print(element + " ");
11.         System.out.println();
12.     }
13. }
```

```
1.  import java.util.*;
2.  class CompDemo2 {
3.      public static void main(String args[]) {
4.          TreeSet<String> ts = new TreeSet<String>((aStr, bStr)
5.                                                   -> bStr.compareTo(aStr));
6.          // Add elements to the tree set.
7.          ts.add("C"); ts.add("A"); ts.add("B");
8.          ts.add("E"); ts.add("F"); ts.add("D");
9.          // Display the elements.
10.         for (String element : ts)
11.             System.out.print(element + " ");
12.         System.out.println();
13.     }
14. }
```

```
1.  import java.util.*;
2.  class TComp implements Comparator<String> {
3.      public int compare(String aStr, String bStr) {
4.          int i, j, k;
5.          // Find index of beginning of last name.
6.          i = aStr.lastIndexOf(' '); j = bStr.lastIndexOf(' ');
7.          k = aStr.substring(i)
8.                  .compareToIgnoreCase(bStr.substring(j));
9.          if (k == 0) // last names match, check entire name
10.             return aStr.compareToIgnoreCase(bStr);
11.         else return k;
12.     }
13. }
```

```
1.  class TreeMapDemo2 {
2.      public static void main(String args[]) {
3.          // Create a tree map.
4.          TreeMap<String, Double> tm = new TreeMap<String, Double>(new TComp());
5.          // Put elements to the map.
6.          tm.put("John Doe", new Double(3434.34));
7.          tm.put("Tom Smith", new Double(123.22));
8.          tm.put("Jane Baker", new Double(1378.00));
9.          tm.put("Tod Hall", new Double(99.22));
10.         tm.put("Ralph Smith", new Double(-19.08));
11.         // Get a set of the entries.
12.         Set<Map.Entry<String, Double>> set = tm.entrySet();
13.
```

```
1.              // Display the elements.
2.              for (Map.Entry<String, Double> me : set) {
3.                      System.out.print(me.getKey() + ": ");
4.                      System.out.println(me.getValue());
5.              }
6.              System.out.println();
7.              // Deposit 1000 into John Doe's account.
8.              double balance = tm.get("John Doe");
9.              tm.put("John Doe", balance + 1000);
10.
11.             System.out.println("John Doe's new balance: " +
12.                             tm.get("John Doe"));
13.     }
14. }
```

```
1.  import java.util.*;
2.  //A comparator that compares last names.
3.  class CompLastNames implements Comparator<String> {
4.      public int compare(String aStr, String bStr) {
5.          int i, j;
6.          // Find index of beginning of last name.
7.          i = aStr.lastIndexOf(' ');
8.          j = bStr.lastIndexOf(' ');
9.          return aStr.substring(i)
10.                     .compareToIgnoreCase(bStr.substring(j));
11.     }
12. }
13.
```

```
1.  import java.util.Comparator;
2.  //Sort by entire name when last names are equal.
3.  class CompThenByFirstName implements Comparator<String> {
4.      public int compare(String aStr, String bStr) {
5.          int i, j;
6.
7.          return aStr.compareToIgnoreCase(bStr);
8.      }
9.  }
```

```
1.  class TreeMapDemo2A {
2.      public static void main(String args[]) {
3.          CompLastNames compLN = new CompLastNames();
4.          Comparator<String> compLastThenFirst = compLN.thenComparing(new
5.          CompThenByFirstName());
6.          TreeMap<String, Double> tm = new TreeMap<String,
7.                                              Double>(compLastThenFirst);
8.          tm.put("John Doe", new Double(3434.34));
9.          tm.put("Tom Smith", new Double(123.22));
10.         tm.put("Jane Baker", new Double(1378.00));
11.         tm.put("Tod Hall", new Double(99.22));
12.         tm.put("Ralph Smith", new Double(-19.08));
13.         Set<Map.Entry<String, Double>> set = tm.entrySet();
```

```
1.              for (Map.Entry<String, Double> me : set) {
2.                  System.out.print(me.getKey() + ": ");
3.                  System.out.println(me.getValue());
4.              }
5.              System.out.println();
6.
7.              double balance = tm.get("John Doe");
8.              tm.put("John Doe", balance + 1000);
9.              System.out.println("John Doe's new balance: " +
10.                                                 tm.get("John Doe"));
11.         }
12.     }
```

The Collection Algorithms

- The Collections Framework defines several algorithms that can be applied to collections and maps.

The Collection Algorithms

Method	Description
static <T> boolean addAll(Collection <? super T> *c*, T... *elements*)	Inserts the elements specified by *elements* into the collection specified by *c*. Returns **true** if the elements were added and **false** otherwise.
static <T> Queue<T> asLifoQueue(Deque<T> *c*)	Returns a last-in, first-out view of *c*.
static <T> int binarySearch(List<? extends T> *list*, T *value*, Comparator<? super T> *c*)	Searches for *value* in *list* ordered according to *c*. Returns the position of value in *list*, or a negative value if *value* is not found.
static <T> int binarySearch(List<? extends Comparable<? super T>> *list*, T *value*)	Searches for *value* in *list*. The list must be sorted. Returns the position of *value* in *list*, or a negative value if *value* is not found.
static <E> Collection<E> checkedCollection(Collection<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> List<E> checkedList(List<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **List**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> Map<K, V> checkedMap(Map<K, V> *c*, Class<K> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **Map**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> NavigableMap<K, V> checkedNavigableMap(NavigableMap<K, V> *nm*, Class<E> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **NavigableMap**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> NavigableSet<E> checkedNavigableSet(NavigableSet<E> *ns*, Class<E> *t*)	Returns a run-time type-safe view of a **NavigableSet**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> Queue<E> checkedQueue(Queue<E> *q*, Class<E> *t*)	Returns a run-time type-safe view of a **Queue**. An attempt to insert an incompatible element will cause a **ClassCastException**.

static <E> List<E> checkedSet(Set<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **Set**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> *c*, Class<K> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **SortedMap**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> SortedSet<E> checkedSortedSet(SortedSet<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **SortedSet**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <T> void copy(List<? super T> *list1*, List<? extends T> *list2*)	Copies the elements of *list2* to *list1*.
static boolean disjoint(Collection<?> *a*, Collection<?> *b*)	Compares the elements in *a* to elements in *b*. Returns **true** if the two collections contain no common elements (i.e., the collections contain disjoint sets of elements). Otherwise, returns **false**.
static <T> Enumeration<T> emptyEnumeration( )	Returns an empty enumeration, which is an enumeration with no elements.
static <T> Iterator<T> emptyIterator( )	Returns an empty iterator, which is an iterator with no elements.
static <T> List<T> emptyList( )	Returns an immutable, empty **List** object of the inferred type.
static <T> ListIterator<T> emptyListIterator( )	Returns an empty list iterator, which is a list iterator that has no elements.
static <K, V> Map<K, V> emptyMap( )	Returns an immutable, empty **Map** object of the inferred type.
static <K, V> NavigableMap<K, V> emptyNavigableMap( )	Returns an immutable, empty **NavigableMap** object of the inferred type.
static <E> NavigableSet<E> emptyNavigableSet( )	Returns an immutable, empty **NavigableSet** object of the inferred type.
static <T> Set<T> emptySet( )	Returns an immutable, empty **Set** object of the inferred type.
static <K, V> SortedMap<K, V> emptySortedMap( )	Returns an immutable, empty **SortedMap** object of the inferred type.
static <E> SortedSet<E> emptySortedSet( )	Returns an immutable, empty **SortedSet** object of the inferred type.
static <T> Enumeration<T> enumeration(Collection<T> *c*)	Returns an enumeration over *c*. (See "The Enumeration Interface," later in this chapter.)
static <T> void fill(List<? super T> *list*, T *obj*)	Assigns *obj* to each element of *list*.

static int frequency(Collection<?> *c*, object *obj*)	Counts the number of occurrences of *obj* in *c* and returns the result.
static int indexOfSubList(List<?> *list*, List<?> *subList*)	Searches *list* for the first occurrence of *subList*. Returns the index of the first match, or –1 if no match is found.
static int lastIndexOfSubList(List<?> *list*, List<?> *subList*)	Searches *list* for the last occurrence of *subList*. Returns the index of the last match, or –1 if no match is found.
static <T> ArrayList<T> list(Enumeration<T> *enum*)	Returns an **ArrayList** that contains the elements of *enum*.
static <T> T max(Collection<? extends T> *c*, Comparator<? super T> *comp*)	Returns the maximum element in *c* as determined by *comp*.
static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> *c*)	Returns the maximum element in *c* as determined by natural ordering. The collection need not be sorted.
static <T> T min(Collection<? extends T> *c*, Comparator<? super T> *comp*)	Returns the minimum element in *c* as determined by *comp*. The collection need not be sorted.
static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> *c*)	Returns the minimum element in *c* as determined by natural ordering.
static <T> List<T> nCopies(int *num*, T *obj*)	Returns *num* copies of *obj* contained in an immutable list. *num* must be greater than or equal to zero.
static <E> Set<E> newSetFromMap(Map<E, Boolean> *m*)	Creates and returns a set backed by the map specified by *m*, which must be empty at the time this method is called.
static <T> boolean replaceAll(List<T> *list*, T *old*, T *new*)	Replaces all occurrences of *old* with *new* in *list*. Returns **true** if at least one replacement occurred. Returns **false** otherwise.
static void reverse(List<T> *list*)	Reverses the sequence in *list*.
static <T> Comparator<T> reverseOrder(Comparator<T> *comp*)	Returns a reverse comparator based on the one passed in *comp*. That is, the returned comparator reverses the outcome of a comparison that uses *comp*.
static <T> Comparator<T> reverseOrder( )	Returns a reverse comparator, which is a comparator that reverses the outcome of a comparison between two elements.
static void rotate(List<T> *list*, int *n*)	Rotates *list* by *n* places to the right. To rotate left, use a negative value for *n*.
static void shuffle(List<T> *list*, Random *r*)	Shuffles (i.e., randomizes) the elements in *list* by using *r* as a source of random numbers.

static void shuffle(List<T> *list*)	Shuffles (i.e., randomizes) the elements in *list*.
static <T> Set<T> singleton(T *obj*)	Returns *obj* as an immutable set. This is an easy way to convert a single object into a set.
static <T> List<T> singletonList(T *obj*)	Returns *obj* as an immutable list. This is an easy way to convert a single object into a list.
static <K, V> Map<K, V> singletonMap(K *k*, V *v*)	Returns the key/value pair *k/v* as an immutable map. This is an easy way to convert a single key/value pair into a map.
static <T> void sort(List<T> *list*, Comparator<? super T> *comp*)	Sorts the elements of *list* as determined by *comp*.
static <T extends Comparable<? super T>> void sort(List<T> *list*)	Sorts the elements of *list* as determined by their natural ordering.
static void swap(List<?> *list*, int *idx1*, int *idx2*)	Exchanges the elements in *list* at the indices specified by *idx1* and *idx2*.
static <T> Collection<T> synchronizedCollection(Collection<T> *c*)	Returns a thread-safe collection backed by *c*.
static <T> List<T> synchronizedList(List<T> *list*)	Returns a thread-safe list backed by *list*.
static <K, V> Map<K, V> synchronizedMap(Map<K, V> *m*)	Returns a thread-safe map backed by *m*.
static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> *nm*)	Returns a synchronized navigable map backed by *nm*.
static <T> NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> *ns*)	Returns a synchronized navigable set backed by *ns*.
static <T> Set<T> synchronizedSet(Set<T> *s*)	Returns a thread-safe set backed by *s*.
static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> *sm*)	Returns a thread-safe sorted map backed by *sm*.
static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> *ss*)	Returns a thread-safe sorted set backed by *ss*.
static <T> Collection<T> unmodifiableCollection(Collection<? extends T> *c*)	Returns an unmodifiable collection backed by *c*.
static <T> List<T> unmodifiableList(List<? extends T> *list*)	Returns an unmodifiable list backed by *list*.
static <K, V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> *m*)	Returns an unmodifiable map backed by *m*.

static <K, V> NavigableMap<K, V> unmodifiableNavigableMap(NavigableMap<K, ? extends V> *nm*)	Returns an unmodifiable navigable map backed by *nm*.
static <T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> *ns*)	Returns an unmodifiable navigable set backed by *ns*.
static <T> Set<T> unmodifiableSet(Set<? extends T> *s*)	Returns an unmodifiable set backed by *s*.
static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> *sm*)	Returns an unmodifiable sorted map backed by *sm*.
static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> *ss*)	Returns an unmodifiable sorted set backed by *ss*.

```
1.  import java.util.*;
2.  class AlgorithmsDemo {
3.      public static void main(String args[]) {
4.          LinkedList<Integer> ll = new LinkedList<Integer>();
5.          ll.add(-8); ll.add(20); ll.add(-20); ll.add(8);
6.          // Create a reverse order comparator.
7.          Comparator<Integer> r = Collections.reverseOrder();
8.          // Sort list by using the comparator.
9.          Collections.sort(ll, r);
10.         System.out.print("List sorted in reverse: ");
11.         for (int i : ll) System.out.print(i + " ");
12.         System.out.println();
```

```
1.              // Shuffle list.
2.              Collections.shuffle(ll);
3.              // Display randomized list.
4.              System.out.print("List shuffled: ");
5.              for (int i : ll) System.out.print(i + " ");
6.              System.out.println();
7.              System.out.println("Minimum: " +
8.                                                  Collections.min(ll));
9.              System.out.println("Maximum: " +
10.                                                 Collections.max(ll));
11.         }
12.     }
```

Arrays

- The Arrays class provides various methods that are useful when working with arrays.
- These methods help bridge the gap between collections and arrays.
- Methods implemented by Arrays can be found here:

https://docs.oracle.com/javase/9/docs/api/java/util/Arrays.html

Example


```
1.  import java.util.*;
2.  class ArraysDemo {
3.      public static void main(String args[]) {
4.          int array[] = new int[10];
5.          for (int i = 0; i < 10; i++) array[i] = -3 * i;
6.          System.out.print("Original contents: ");
7.          display(array);
8.          Arrays.sort(array);
9.          System.out.print("Sorted: ");
10.         display(array);
11.         // Fill and display the array.
12.         Arrays.fill(array, 2, 6, -1);
13.         System.out.print("After fill(): ");
14.         display(array);
```

Example


```
1.              Arrays.sort(array);
2.              System.out.print("After sorting again: ");
3.              display(array);
4.              System.out.print("The value -9 is at location ");
5.              int index = Arrays.binarySearch(array, -9);
6.              System.out.println(index);
7.          }
8.          static void display(int array[]) {
9.              for (int i : array)
10.                 System.out.print(i + " ");
11.             System.out.println();
12.         }
13.     }
```

QUESTION ?