

CS 251

Intermediate Programming

Collection Interfaces

Brooke Chenoweth

University of New Mexico

Fall 2025

Collection interface

- The Collection interface lets you pass around a group of objects in the most general way. It could be a List, a Set, or any other kind of Collection.
- Most general purpose collection implementations have a *conversion constructor* that takes a Collection argument. (Constructors aren't part of an interface, but this is a common convention.)

Collection methods

- size, isEmpty
- contains
- add, addAll
- remove, removeAll, retainAll, clear
- toArray
- iterator (because Collection implements Iterable)

Optional methods that are not supported by a specific implementation throw

`UnsupportedOperationException`

Traversing a collection – using for-each

```
public static <T> void printCollection(  
    Collection<T> collection) {  
    for (T obj : collection) {  
        System.out.println(obj);  
    }  
}
```

In this example, the *generic method* has a type parameter `T`, but we don't really use it for anything. Instead, we could use a *wildcard* type

Traversing a collection – using for-each

```
public static void printCollection(  
    Collection<?> collection) {  
    for (Object obj : collection) {  
        System.out.println(obj);  
    }  
}
```

Using a ? for the type parameter is a wildcard indicating an unknown type.

Traversing a collection – using iterator

```
public static void printWithIterator(  
    Collection<?> collection) {  
    Iterator<?> iter = collection.iterator();  
    while(iter.hasNext()) {  
        System.out.println(iter.next());  
    }  
}
```

Iterator and Iterable

Iterator has three methods

- hasNext – Are there more elements?
- next – Return the next element
- remove – Remove the last element returned (optional)

Iterable only has one method

- iterator – returns an Iterator
- Implementing this interface allows object to be target of for-each

toArray

Sometimes we have to work with an older API that is expecting arrays.

```
Collection<String> strColl;  
// ... actually initialize the collection here...  
  
// No argument version gives an array of Objects  
Object[] objArr = strColl.toArray();  
  
// Here we tell it that we want a String array  
String[] strArr = strColl.toArray(new String[0]);
```

Set

A Set implements only the methods inherited from Collection, but adds the additional restriction that duplicate elements are prohibited.

- HashSet – fast, offers no guarantee about order
- TreeSet – implements SortedSet, uses *natural ordering* or a Comparator

Beware of placing mutable items in the set!

List

- add, addAll – add to end of the list
- remove – removes first occurrence
- iterator, listIterator
- indexOf, lastIndexOf – find index of element
- get, set – access element at given index
- subList – view portion of list as a List

Making a list of arguments

```
public static void main(String[] args) {  
    List<String> strList = new ArrayList<String>();  
    for(String arg : args) {  
        strList.add(arg);  
    }  
  
    // Let's sort the arguments  
    Collections.sort(strList);  
    System.out.println(strList);  
}
```

Using Arrays.asList

```
public static void main(String[] args) {  
    List<String> strList = Arrays.asList(args);  
  
    // Let's shuffle this time.  
    Collections.shuffle(strList);  
    System.out.println(strList);  
}
```

ListIterator

ListIterator extends Iterator interface with additional functionality

- hasNext, next, remove – inherited from parent
- hasPrevious, previous – allow traversal in reverse
- nextIndex, previousIndex – get index of element
- add – insert element in list
- set – replace element

List Algorithms

Most polymorphic algorithms in `Collections` class apply to `List`.

- `sort` – sorts a `List` using a fast, stable sort.
- `shuffle` – randomly permutes the elements
- `reverse` – reverses the order of the elements
- `rotate` – rotates all the elements in a `List` by a specified distance.
- `swap` – swaps the elements at specified positions in a `List`.
- `replaceAll` – replaces all occurrences of one specified value with another.

List Algorithms

- fill – overwrites every element in a List with the specified value.
- copy – copies the source List into the destination List.
- binarySearch – searches for an element in an ordered List using the binary search algorithm.
- indexOfSubList – returns the index of the first sublist of one List that is equal to another.
- lastIndexOfSubList – returns the index of the last sublist of one List that is equal to another.

Queue

- Insert – add, offer
- Remove – remove, poll
- Examine – element, peek

Queues usually use FIFO order. PriorityQueue will use natural ordering or a Comparator.

Deque

Double ended queue, can insert and remove at both ends. Implements both stack and queue at same time.

	Beginning	End
Insert	addFirst, offerFirst	addLast, offerLast
Remove	removeFirst, pollFirst	removeLast, pollLast
Examine	getFirst, peekFirst	getLast, peekLast

Map

Maps keys to values. Does not implement `Collection` itself, but has three *collection views*

- `keySet` – Set of the keys
- `values` – Collection of values
- `entrySet` – Set of key-value mappings.

Beware of using mutable objects as keys!

Counting word frequency

```
public static void main(String[] args) {  
    Map<String, Integer> wordCountMap =  
        new HashMap<String, Integer>();  
  
    for(String arg : args) {  
        Integer count = wordCountMap.get(arg);  
        if(count == null) {  
            count = 0;  
        }  
        count++;  
        wordCountMap.put(arg, count);  
    }  
  
    System.out.println(wordCountMap.size() + " words");  
    System.out.println(wordCountMap);  
}
```

Comparable interface

```
// T is type of objects that this  
// object may be compared to  
public interface Comparable<T>{  
  
    // Compares this with other object  
    // Returns negative, zero, or positive integer when  
    // less than, equal to, or greater than other  
    int compareTo(T o);  
}
```

Comparable Names

```
public class Name implements Comparable<Name> {
    private final String firstName, lastName;

    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public int compareTo(Name n) {
        int result = lastName.compareTo(n.lastName);
        if(result == 0) {
            result = firstName.compareTo(n.firstName);
        }
        return result;
    }

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
```

Comparator interface

```
// T is type of objects to compare
public interface Comparator<T>{

    // Compares two objects
    // Returns negative, zero, or positive integer when
    // o1 is less than, equal to, or greater than o2
    int compare(T o1, T o2);
}
```

Sorting by first name

```
Comparator<Name> firstNameOrder =  
    new Comparator<Name>() {  
        public int compare(Name n1, Name n2) {  
            String first1 = n1.getFirstName();  
            String first2 = n2.getFirstName();  
            int result = first1.compareTo(first2);  
            if(result == 0) {  
                String last1 = n1.getLastName();  
                String last2 = n2.getLastName();  
                result = last1.compareTo(last2);  
            }  
            return result;  
        }  
    };  
};
```

```
List<Name> names = new ArrayList<Name>();  
// Add some names here...
```

```
Collections.sort(names, firstNameOrder);
```