# CENG 1004
# Introduction to Object Oriented Programming

Spring 2018

# WEEK 10 - 11

# Collections

# Java Collections Framework

A ***collection*** is a container object that represents a group of objects, which are referred to as *elements*.

The Java Collections Framework supports three types of collections:

*sets, queues*

*lists,* and

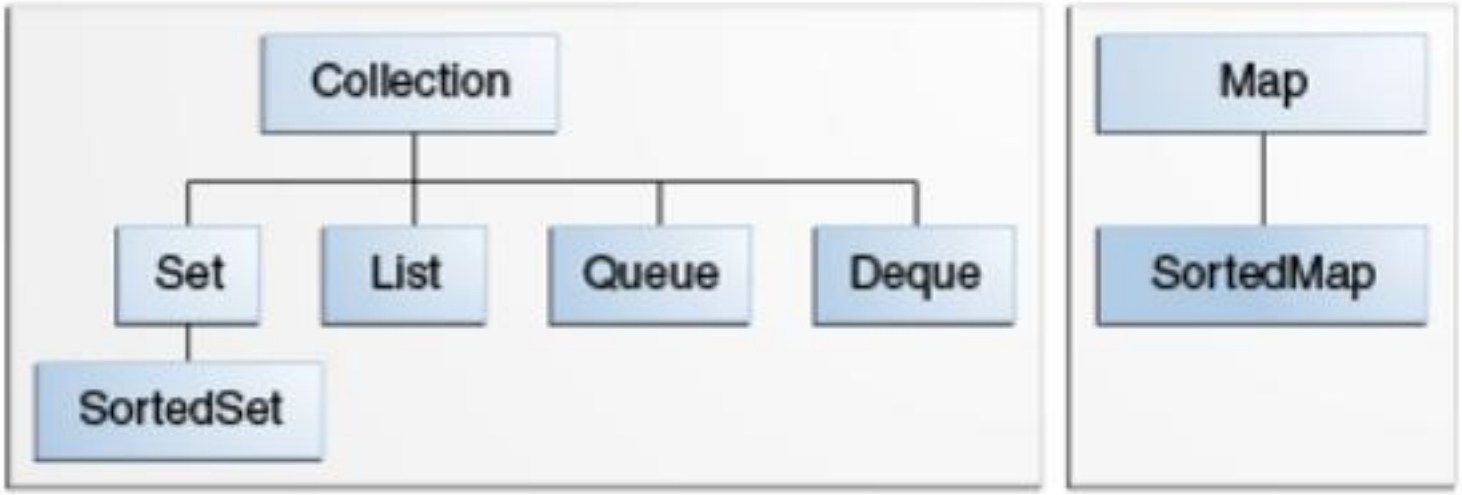
*maps*.

# Java Collections Framework

- All collections frameworks contain the following:
  - **Interfaces:** abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation.
  - **Implementations:** These are the concrete implementations of the collection interfaces.
  - **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

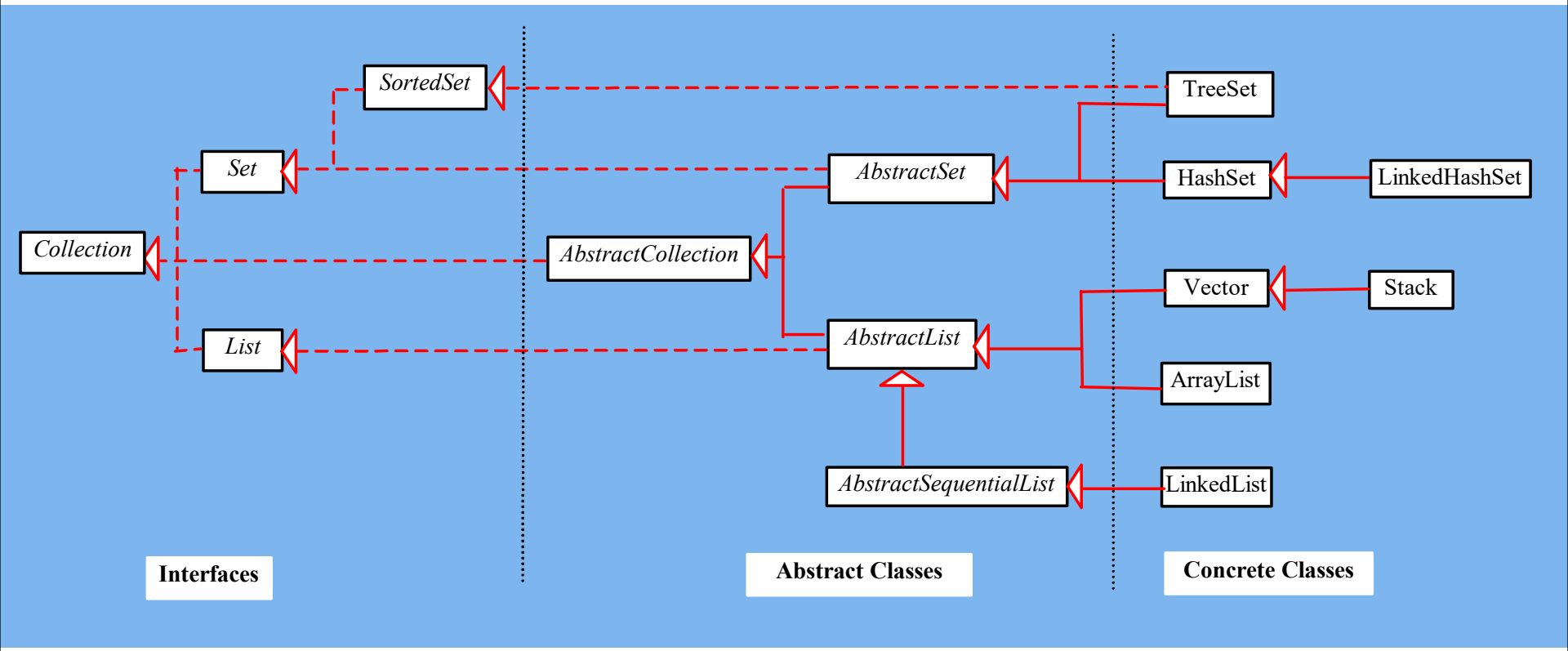
# Collection Interfaces

- Allow collections to be manipulated independently of the details of their representation.



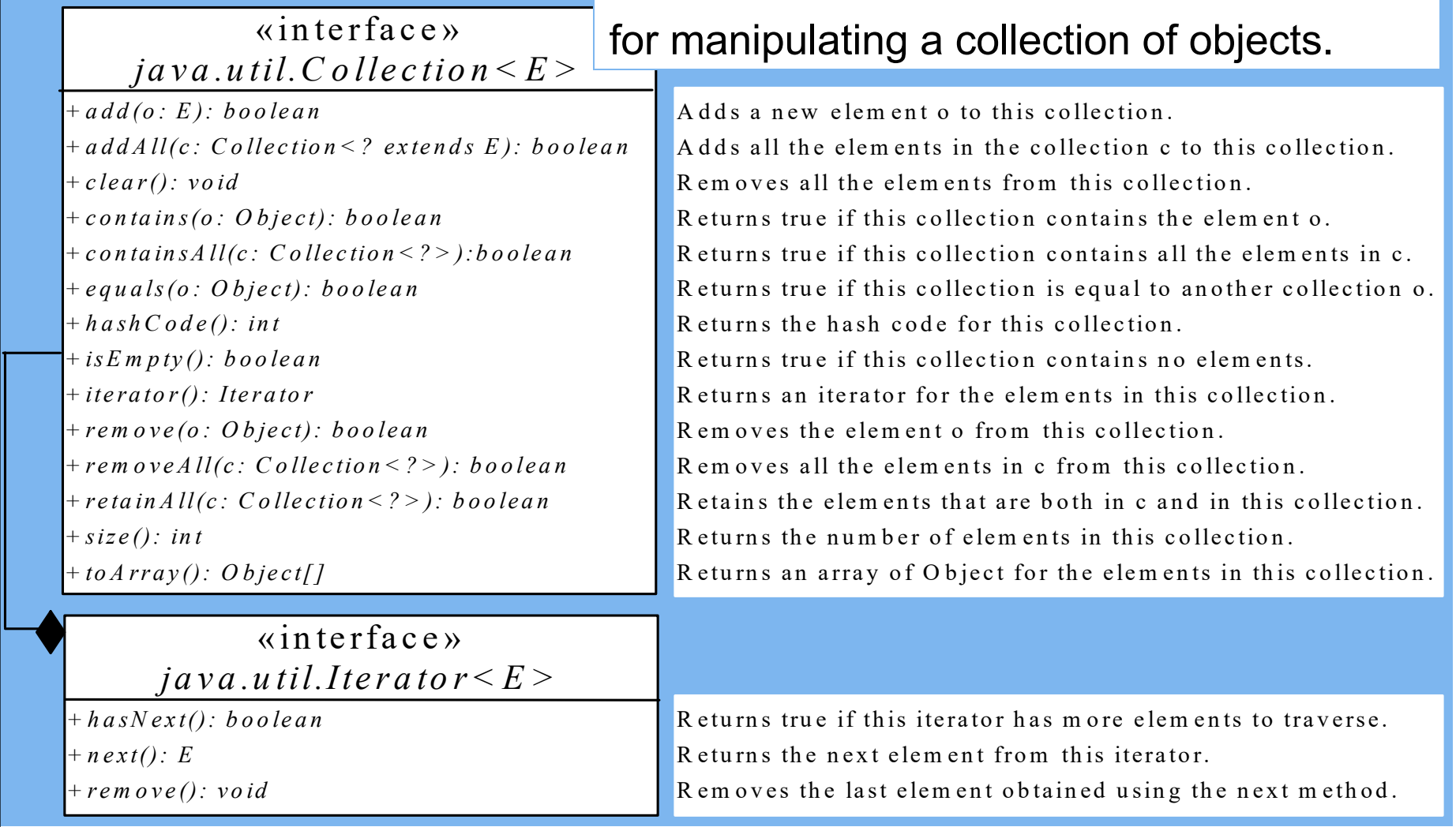
# Java Collection Framework hierarchy, cont.

*Set and List are subinterfaces of Collection.*



# The Collection Interface

The Collection interface is the root interface for manipulating a collection of objects.

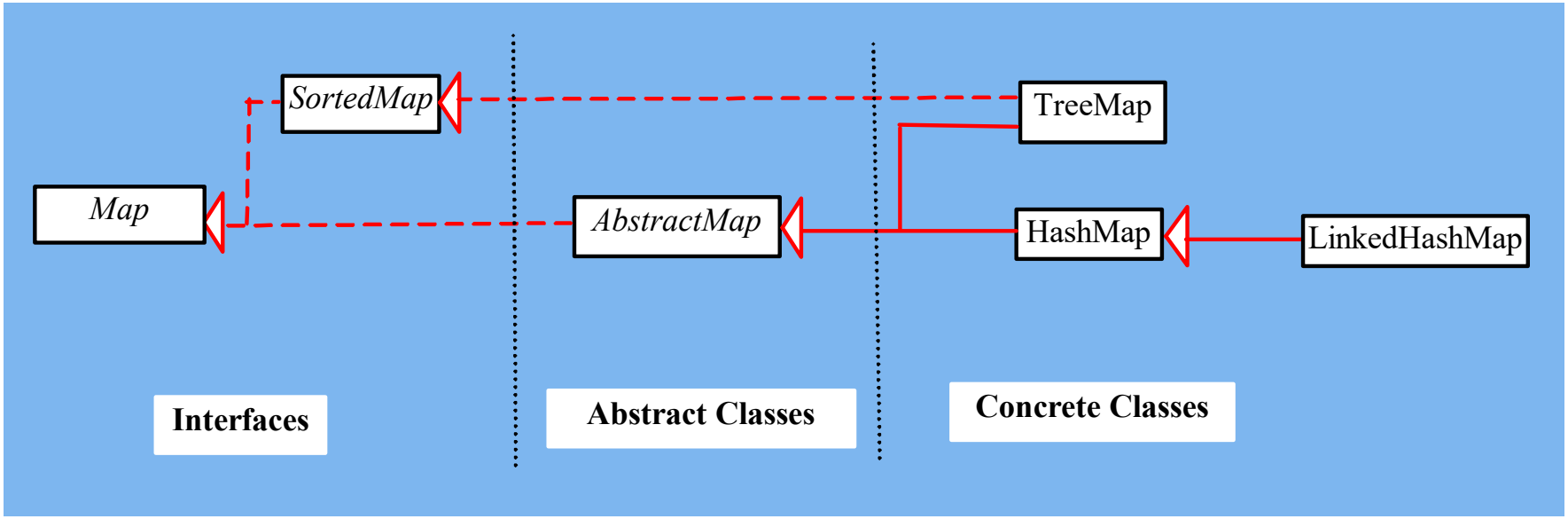


# Java Collection Framework hierarchy, cont.

An instance of Map represents a group of objects, each of which is associated with a key.

-- Use a key to get the object from a map, and

-- Must use a key to put the object into the map.



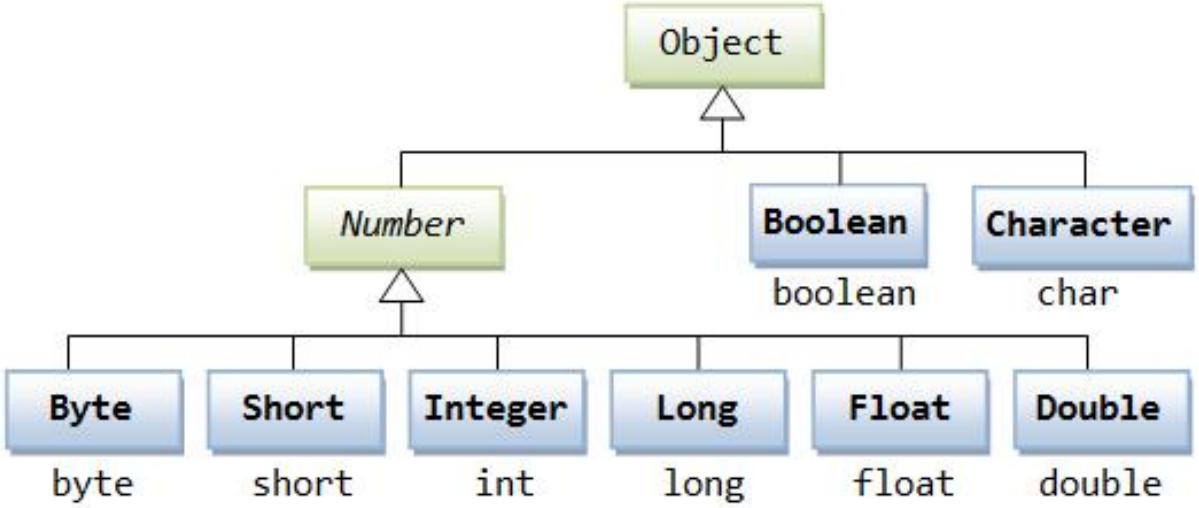
# The Collection Interface

- The Collection interface is used to pass around collections of objects where maximum generality is desired.

- By convention, all general-purpose collection implementations have a constructor that takes a Collection argument.

# Auto-Boxing & Auto-Unboxing

- A collection contains only objects.
- To put a primitive into a collection (such as ArrayList), you have to wrap the primitive into an object using the corresponding wrapper class as shown below:



# Auto-Boxing & Auto-Unboxing

- Prior to JDK 1.5, you have to wrap a primitive value into an object and unwrap the primitive value from the wrapper object:

```
// Pre-JDK 1.5
Integer intObj = new Integer(5566);    // wrap int to Integer
int i = intObj.intValue();             // unwrap Integer to int

Double doubleObj = new Double(55.66);  // wrap double to Double
double d = doubleObj.doubleValue();    // unwrap Double to double
```

- JDK 1.5 introduces a new feature called auto-boxing and auto-unboxing to resolve this problem, by delegating the compiler to do the job.

```
// JDK 1.5
Integer intObj = 5566;    // autobox from int to Integer
int i = intObj;           // auto-unbox from Integer to int

Double doubleObj = 55.66; // autoboxing from double to Double
double d = doubleObj;     // auto-unbox from Double to double
```

# Auto-Boxing & Auto-Unboxing

- Auto-Boxing & Auto-Unboxing are used when you add primitive values or get those values to/from collections:

```
List<Integer> lst = new ArrayList<Integer>();

lst.add(7); // autobox to Integer

// Retrieve via for-loop with List's index
for (int i = 0; i < lst.size(); ++i) {
    int j = lst.get(i);   // downcast to Integer, auto-unbox to int
    System.out.println(j);
}
```

# Auto-Boxing & Auto-Unboxing

- What is the expected output of following code?

```
List<Integer> lst = new ArrayList<>();
lst.add(1);
lst.add(2);
lst.add(3);

lst.remove(2);

System.out.println(lst);
```

# Auto-Boxing & Auto-Unboxing

- The output will be [1,2]

```
Collection Interface has boolean remove(Object o);

List Interface introduces E remove(int index);
```

# Traversing Collections

- There are several ways to traverse collections:
  - for-each Construct

```
for (Object o : col)
        System.out.println(o)
```

  - Stream API
    - col.stream().forEach(e -> System.out.println(e));

# Traversing Collections

– Iterators

```
Iterator<String> itr = col.iterator();
while (itr.hasNext())
        System.out.println(itr.next());
```

– Index based looping (List)

```
//index based looping
for (int i = 0; i<col.size(); i++){
        System.out.println(((List)col).get(i));
}
```

# Removing while Traversing

- What will be the content of the list after executing the below code given that list contains [5, 7, 11, 12, 21]

```
for(int i=0; i< list.size(); i++){

   int element = list.get(i);

   //delete numbers between 10 and 20
   if ((element>=10)&&(element<20)){
       list.remove(i);
   }
}
```

# Removing while Traversing

- The resulting list will be [5, 7, 12, 21]
- If we remove the element at the second index, "12" will be moved to second index and 21 will be moved to third index
- You should be careful if you remove or add elements to the list while traversing it.

# Removing while Traversing

- What is wrong with this code?

```
for (int element: list){
   System.out.println("Checking element " + element);
   if ((element>=10)&&(element<20)){
       list.remove(element);
   }
}
```

# Removing while Traversing

- What is wrong with this code?

```
for (int element: list){
   System.out.println("Checking element " + element);
   if ((element>=10)&&(element<20)){
       list.remove(element);
   }
}
```

**Collection Interface has `boolean remove(Object o);`**

**List Interface introduces `E remove(int index);`**

# Removing while Traversing

- Corrected code will give another error

```
for (Integer element: list){
   System.out.println("Checking element " + element);
   if ((element>=10)&&(element<20)){
       list.remove(element);
   }
}
Exception in thread "main"
   java.util.ConcurrentModificationException
```

- You can not modify a collection while traversing it with for-each construct

# Removing while Traversing

- **An Iterator** is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.

```
Iterator<Integer> itr = list.iterator();
while (itr.hasNext()){
   int element = itr.next();
   if ((element>=10)&&(element<20)){
       itr.remove();
   }
}
```

- Iterator.remove is the only safe way to modify a collection during iteration;

# Collection Interface Bulk Operations

- **containsAll** — returns true if the target Collection contains all of the elements in the specified Collection.
- **addAll** — adds all of the elements in the specified Collection to the target Collection.
- **removeAll** — removes from the target Collection all of its elements that are also contained in the specified Collection.

# Collection Interface Bulk Operations

- **retainAll** — removes from the target Collection all its elements that are not also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.
- **clear** — removes all elements from the Collection.

# Collection Interface Bulk Operations

- Suppose you want to remove all of the null elements from a Collection.

```
c.removeAll(Collections.singleton(null));
```

- Collections.singleton, which is a static factory method that returns an immutable Set containing only the specified element.

# Collection Interface Array Operations

- The toArray methods are provided as a bridge between collections and older APIs that expect arrays on input.

```
Object[] a = col.toArray();
```

- The following snippet dumps the contents of c into a newly allocated array of String whose length is identical to the number of elements in c.

```
String[] a = c.toArray(new String[0]);
```

# The Set Interface

**Set interface** extends the Collection interface.

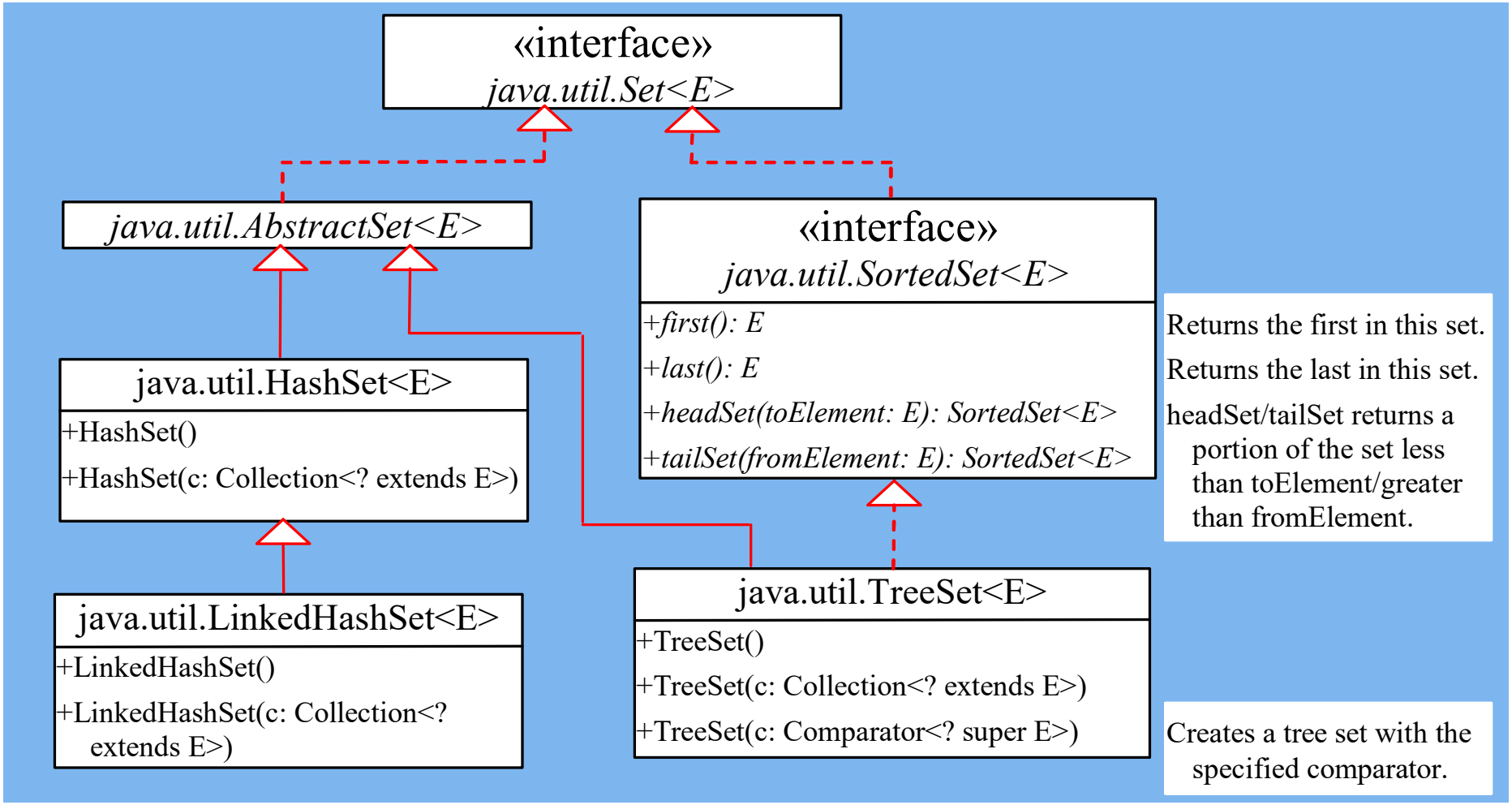
-- Set does not introduce new methods or constants

-- A Set instance must not contains duplicate elements.

--Any concrete classes that implement Set must ensure that no duplicate elements can be added to the set.

-- No two elements e1 and e2 can be in the set such that e1.equals(e2) is true.

# The Set Interface Hierarchy



# The AbstractSet Class

AbstractSet class

- extends AbstractCollection and implements Set.

- Implements the equals() and hashCode() methods

- Hash code of a set == the sum of the hash code of all the elements in the set.

- size() and iterator () methods are not implemented

- Hence AbstractSet is an abstract class.

# The HashSet Class

HashSet class

- Concrete class that implements Set
  -Set and Hash
- Use: to store duplicate-free elements
- For efficiency, objects added to a hash set need to implement the **hashCode** method in a manner that properly distributes the hash code.

# Example: Using HashSet and Iterator

```
//creates a hash set filled with strings
 // Create a hash set
Set set = new HashSet();

// Text in String
 String text = "Have a good day. Have a good class. " +
"Have a good visit. Have fun!“;

// Extract words from text
  StringTokenizer st = new StringTokenizer(text, " .!?");
  while (st.hasMoreTokens())
      set.add(st.nextToken());
```

# Display Set

// First way to display set

```
System.out.println(set);
```

/*Second way to display set: iterator to traverse the elements in the set */

```
// Obtain an iterator for the hash set
Iterator iterator = set.iterator();

// Display the elements in the hash set
while (iterator.hasNext()) {
  System.out.print(iterator.next() + " ");
}
```

# Third way to display set

/**Third way to display set*/

```
for (Object element: set)
      System.out.print(element + " ");

System.out.println("\n");
```

//For-each construct hides the iterator

// Initial string

```
String text = "Have a good day. Have a good class.
   " + "Have a good visit. Have fun!";
```

// Set action: drop duplicates

```
[Have, a, good, day, class, visit, fun]
```

// Program prints: See the hash effect

```
First way:
[a, Have, visit, good, day, class, fun]

Second way:
a Have visit good day class fun

Third way:
a Have visit good day class fun
```

# LinkedHashSet

Properties:

- Set – all elements unique
- Hash – fast access
- Linked – **entry order preserved.**

# Using LinkedHashSet – entry order preserved.

```
// Create a linked hash set
    Set set = new LinkedHashSet();
// Initial string
String text = "Have a good day. Have a good class. "
  + "Have a good visit. Have fun!";
// Set action – drop duplicates
[Have, a, good, day, class, visit, fun]
// Print
[Have, a, good, day, class, visit, fun]
Have a good day class visit fun
```

# SortedSet Interface and TreeSet Class

**SortedSet** is a subinterface of Set

- elements in the set must be sorted.

**TreeSet** is a concrete class that implements the SortedSet interface.

- Use an iterator to traverse the elements in the sorted order.
- The elements can be sorted in two ways.

# Two ways to sort elements in TreeSet Class

One way: Use the ***Comparable*** interface.

Second way: *order by* ***comparator***.

- Specify a comparator for the elements in the set

-if the class for the elements does not implement the Comparable interface, or

- you don't want to use the compareTo method in the class that implements the Comparable interface.

# Comparable interface

```
public interface Comparable<T> {
   public int compareTo(T o);
}
```

- Compares this object with the specified object for order.

- Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

# Example:

Using TreeSet to Sort Elements in a Set

1. Creates a hash set filled with strings

```
1.  Set<String> set = new HashSet<String>();
```

2. Creates a tree set for the same strings

```
2.  TreeSet<String> treeSet
3.          = new TreeSet<String>(set);
```

3. Sort the strings in the tree set using the compareTo method in the Comparable interface

3. How? See the TreeSort constructors in documentation (next slide).

# **TreeSet**(Collection<? extends E> c)

- **TreeSet**
- public **TreeSet**(Collection<? extends E> c) Constructs a new tree set containing the elements in the specified collection, sorted according to the *natural ordering* of its elements. All elements inserted into the set must implement the Comparable interface. Furthermore, all such elements must be *mutually comparable*: e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the set.

- **Parameters:**c - collection whose elements will comprise the new set
- **Throws:** ClassCastException - if the elements in c are not Comparable, or are not mutually comparable NullPointerException - if the specified collection is null

# Output from TreeSort

## // Initial string

```
String text = "Have a good day. Have a good class. " +
"Have a good visit. Have fun!";
```

## // Set action – drop duplicates

```
[Have, a, good, day, class, visit, fun]
```

## // TreeSort action implicitly

```
[Have, a, class, day, fun, good, visit]

Have a class day fun good visit

Have a class day fun good visit
```

# TreeSet constructor with Comparator parameter

- **TreeSet**
- public **TreeSet**(Comparator<? super E> comparator) Constructs a new, empty tree set, sorted according to the specified comparator. All elements inserted into the set must be *mutually comparable* by the specified comparator: comparator.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the set. If the user attempts to add an element to the set that violates this constraint, the add call will throw a ClassCastException.
- **Parameters:**comparator - the comparator that will be used to order this set. If null, the natural ordering of the elements will be used.

# The Comparator Interface

In case:

- you want to insert elements of different types into a tree set.

- The elements may not be instances of Comparable or are not comparable.

You can define a comparator to compare these elements.

create a class that implements the java.util.Comparator interface. The Comparator interface has two methods, compare and equals.

# The Comparator Interface

public int compare(Object element1, Object element2)

Returns a negative value if element1 is less than element2, a positive value if element1 is greater than element2, and zero if they are equal.

public boolean equals(Object element)

Returns true if the specified object is also a comparator and imposes the same ordering as this comparator.

# sort elements in TreeSet

- Comparable example

```
public class Student implements Comparable {

 @Override
    public int compareTo(Student o2) {
    // descending order (ascending order would be:
        // getGrade()-o2.getGrade())
        return getGrade() - o1.getGrade();
    }
}
```

- Comparator example

```
import java.util.Comparator;

public class GradeComparator implements Comparator {
    @Override
    public int compare(Student o1, Student o2) {
        // descending order (ascending order would be:
        // o1.getGrade()-o2.getGrade())
        return o2.getGrade() - o1.getGrade();
    }
}
```

# The List Interface

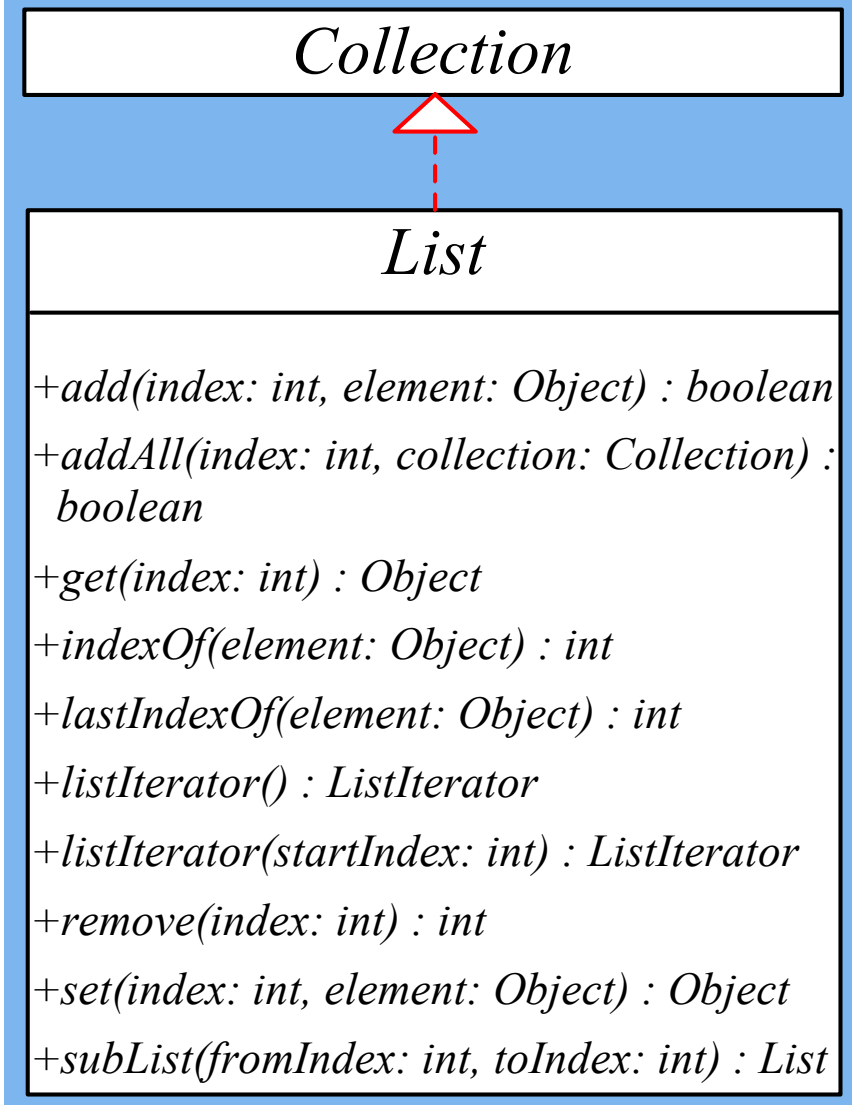
A set stores non-duplicate elements.

Use a list to allow duplicate elements to be stored in a collection

A list can:

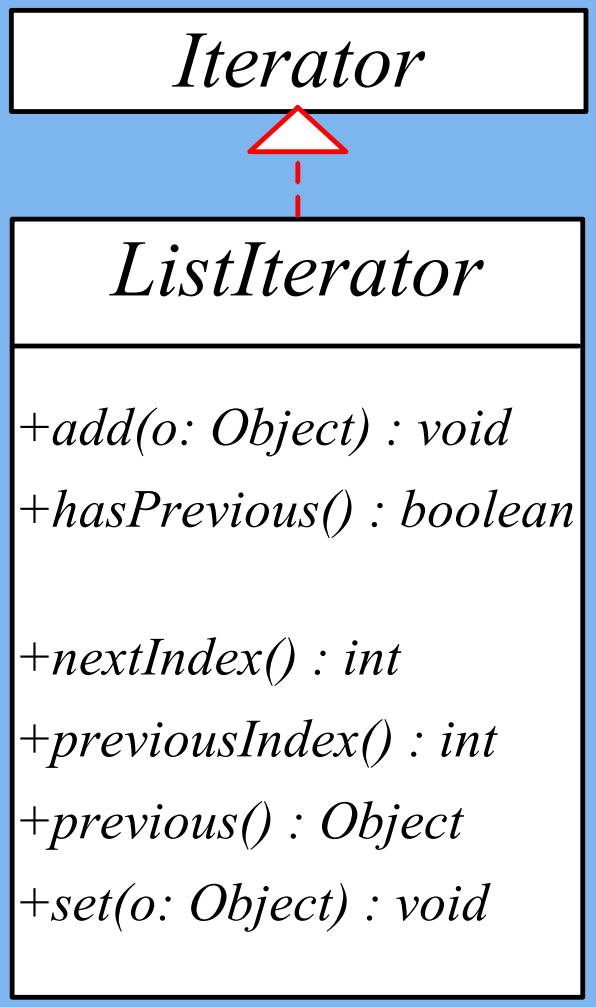
- store duplicate elements,
- allow the user to specify where the element is stored
- The user can access the element by index.

# The List Interface without Generics, cont.




| | |
|---|---|
| Adds a new element at the specified index | |
| Adds all elements in the collection to this list at the specified index | |
| Returns the element in this list at the specified index | |
| Returns the index of the first matching element | |
| Returns the index of the last matching element | |
| Returns the list iterator for the elements in this list | |
| Returns the iterator for the elements from startIndex | |
| Removes the element at the specified index | |
| Sets the element at the specified index | |
| Returns a sublist from fromIndex to toIndex | |

# The List Iterator without Generics




Adds the specified object to the list

Returns true if this list iterator has more elements when traversing backward.

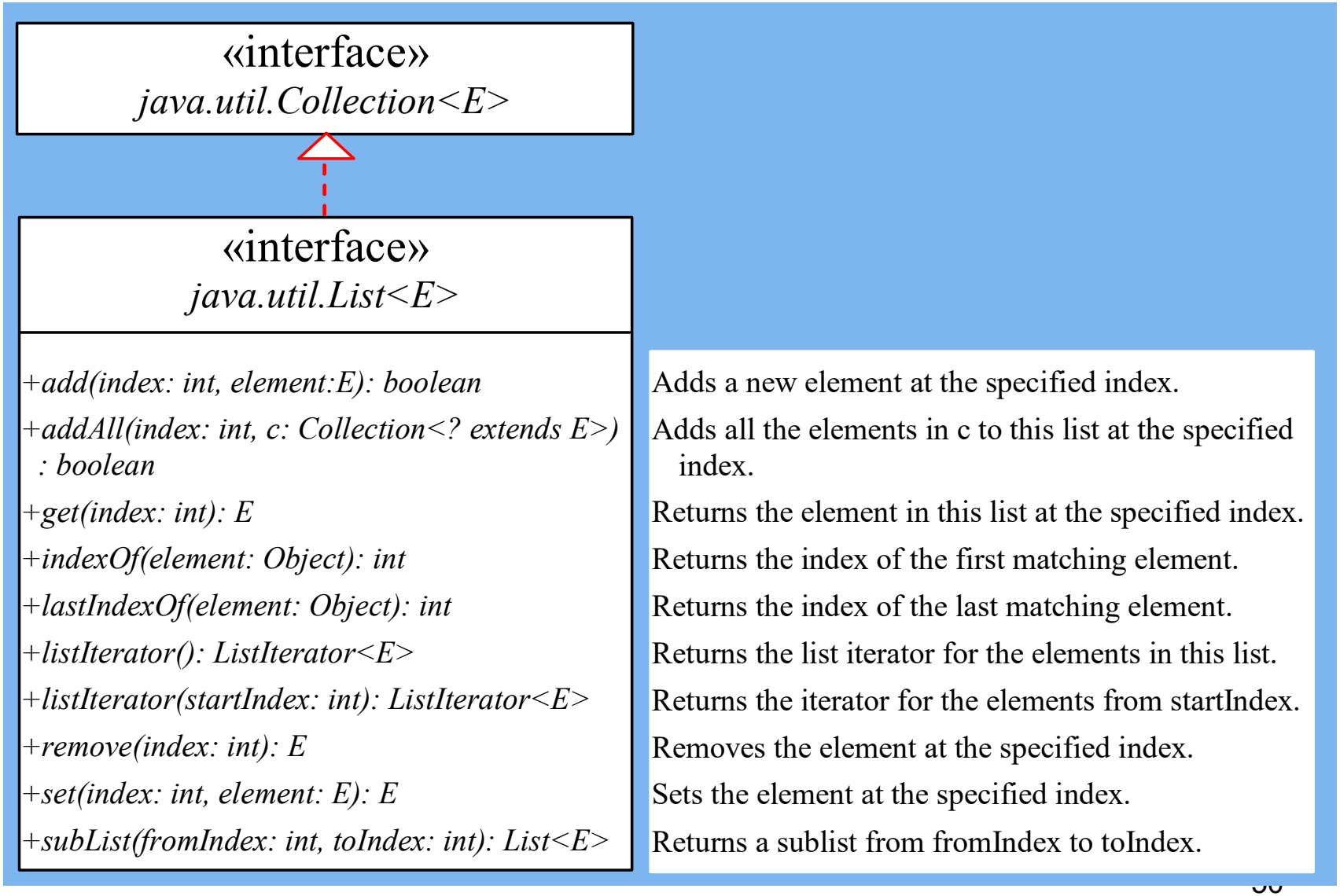
Returns the index of the next element

Returns the index of the previosu element

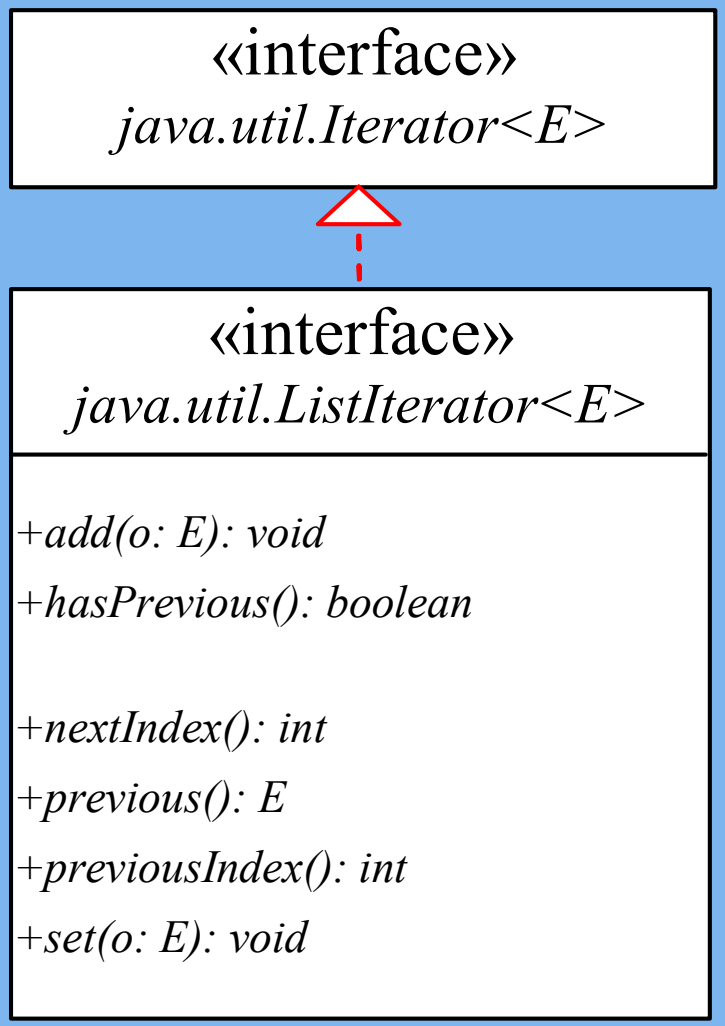
Returns the previous element in this list iterator

Replaces the last element returned by the previous or next method with the specified element

# List Interface with Generics



# ListIterator Interface with Generics




Adds the specified object to the list.

Returns true if this list iterator has more elements when traversing backward.

Returns the index of the next element.

Returns the previous element in this list iterator.

Returns the index of the previous element.

Replaces the last element returned by the previous or next method with the specified element.

# ArrayList and LinkedList

ArrayList class and the LinkedList class:

-- concrete implementations of the List interface.

-- use depends on specific needs

ArrayList most efficient if:

- need to support random access through an index
- without inserting or removing elements from any place other than the end,

LinkedList if:

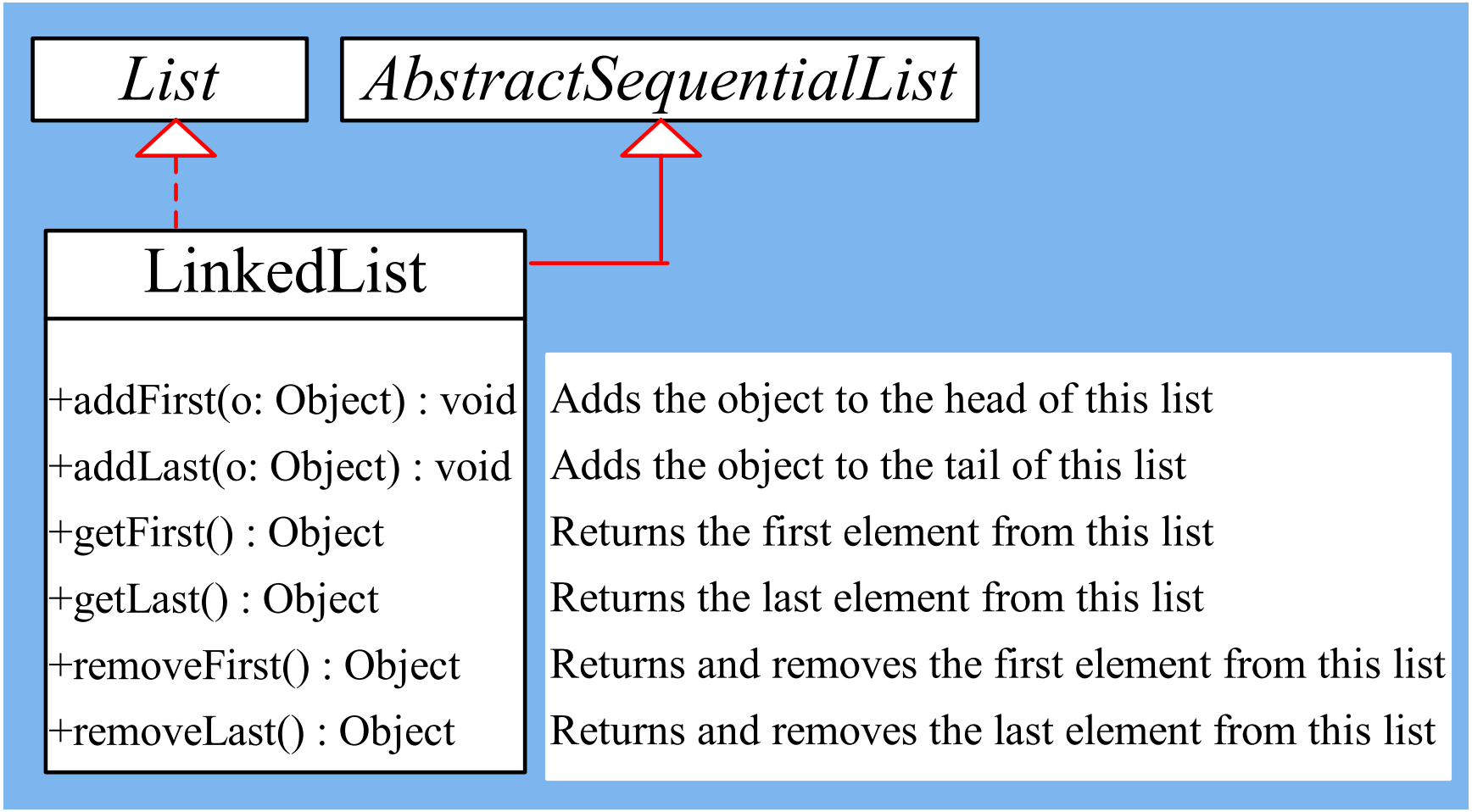
- application requires insertion or deletion of elements from any place in the list

# List versus Array

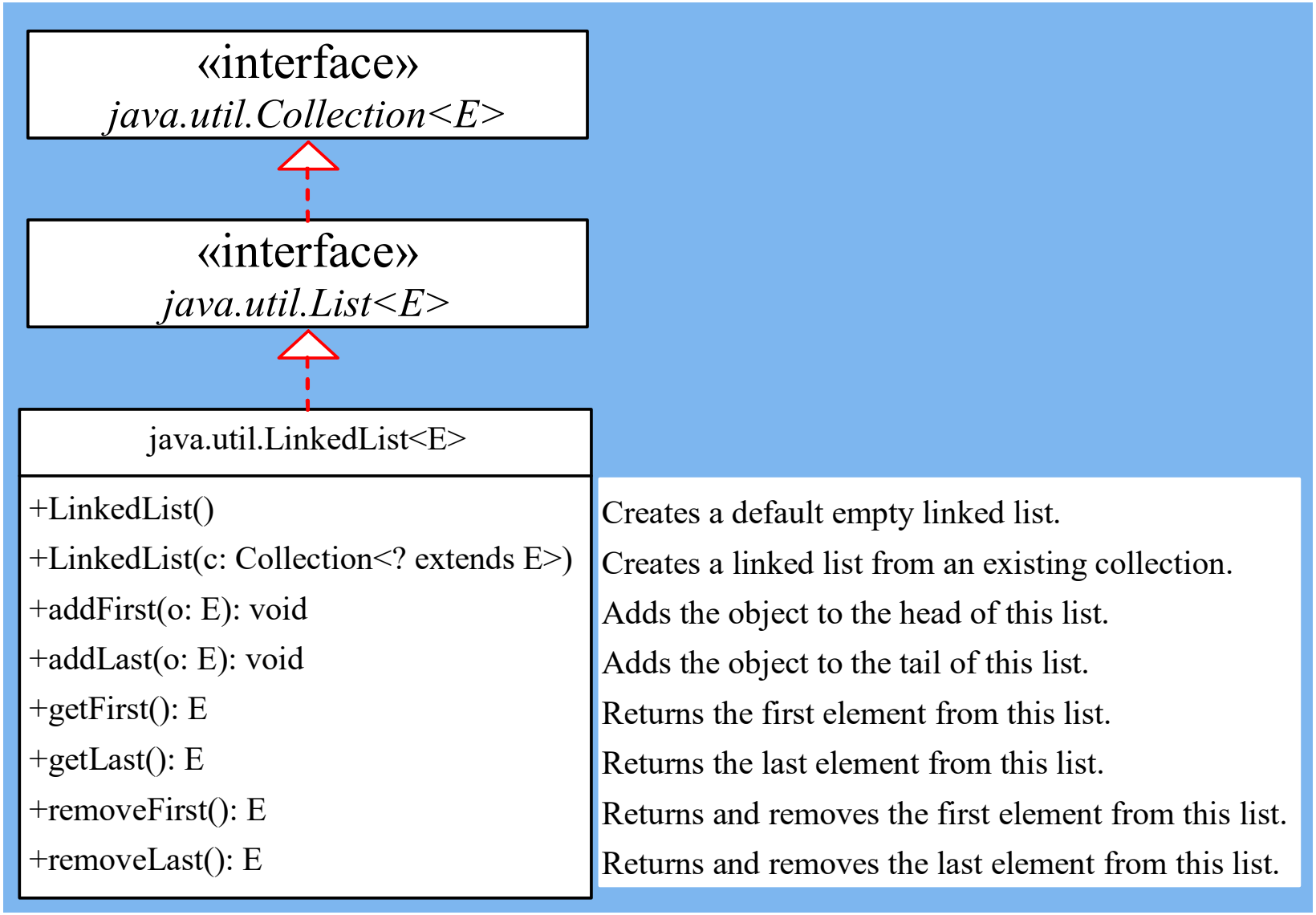
- A list can grow or shrink dynamically.
- An array is fixed once it is created.

- If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

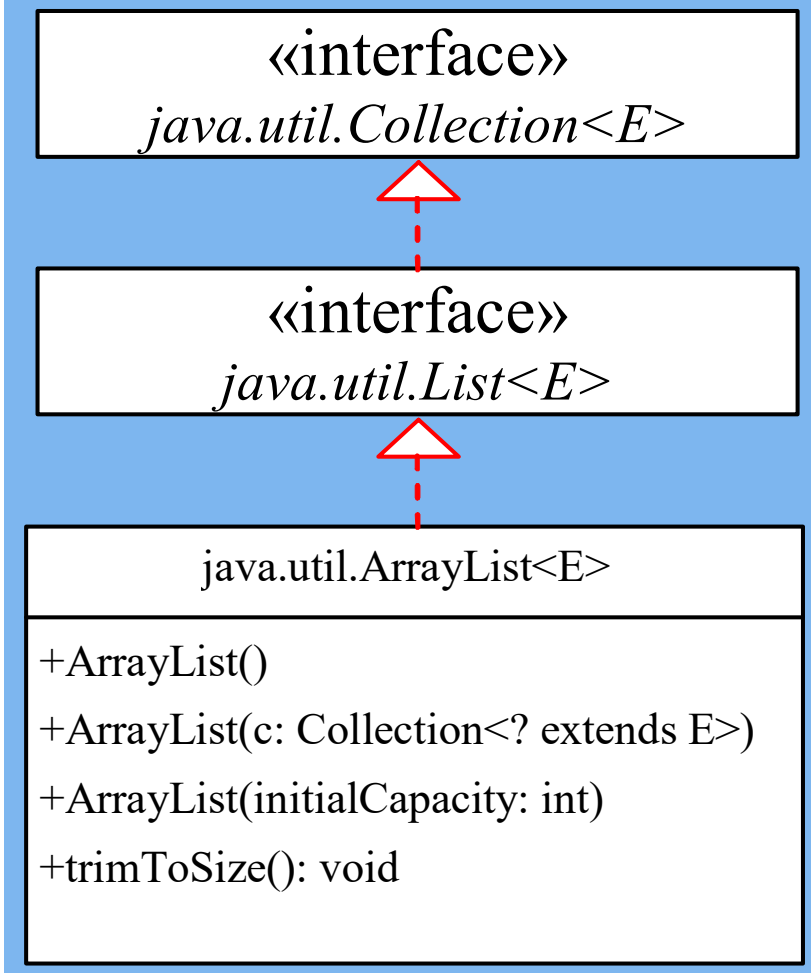
# LinkedList without Generics



# LinkedList with Generics



# ArrayList with Generics




Creates an empty list with the default initial capacity.

Creates an array list from an existing collection.

Creates an empty list with the specified initial capacity.

Trims the capacity of this ArrayList instance to be the list's current size.

# The Vector and Stack Classes

- Java Collections Framework introduced with Java 2.
- Several data structures were supported prior to Java 2.
- Vector class and the Stack class.
- redesigned to fit into the Java Collections Framework
- old-style methods are retained for compatibility.

# The Vector Class

- Java 2 Vector is the same as ArrayList

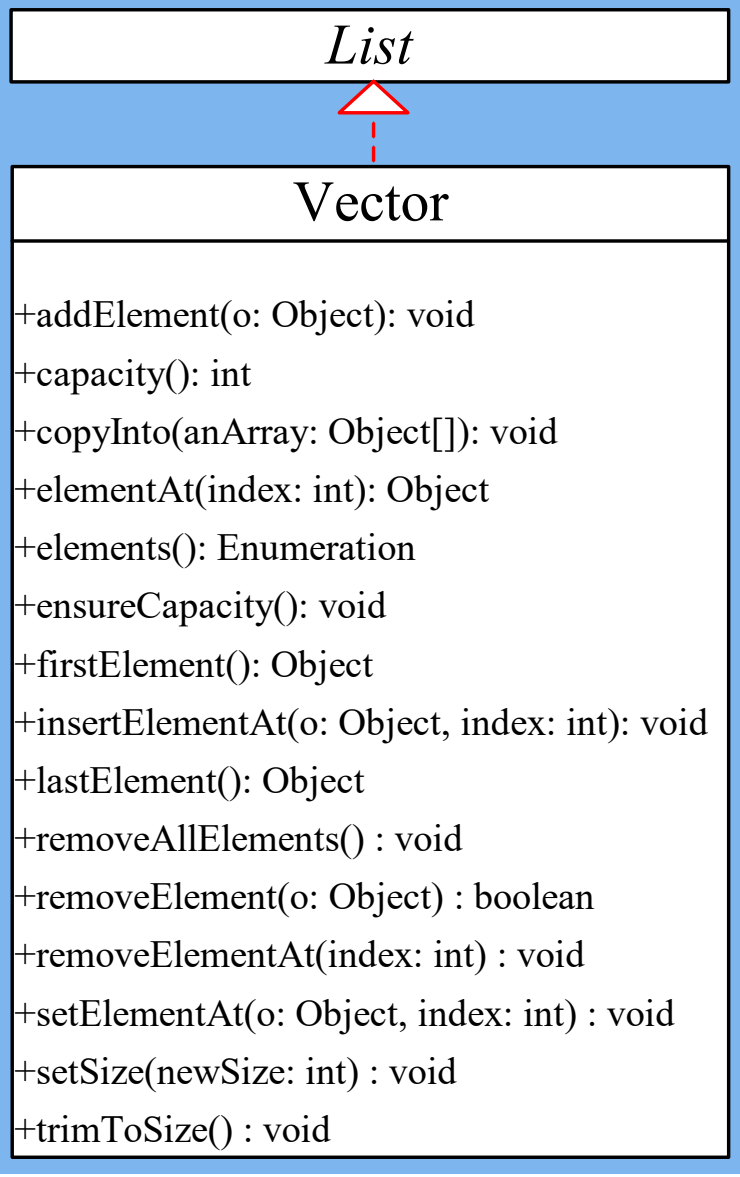
except that Vector contains the synchronized methods for accessing and modifying the vector.

- None of the new collection data structures are synchronized.

- If synchronization is required, you can use the synchronized versions of the collection classes

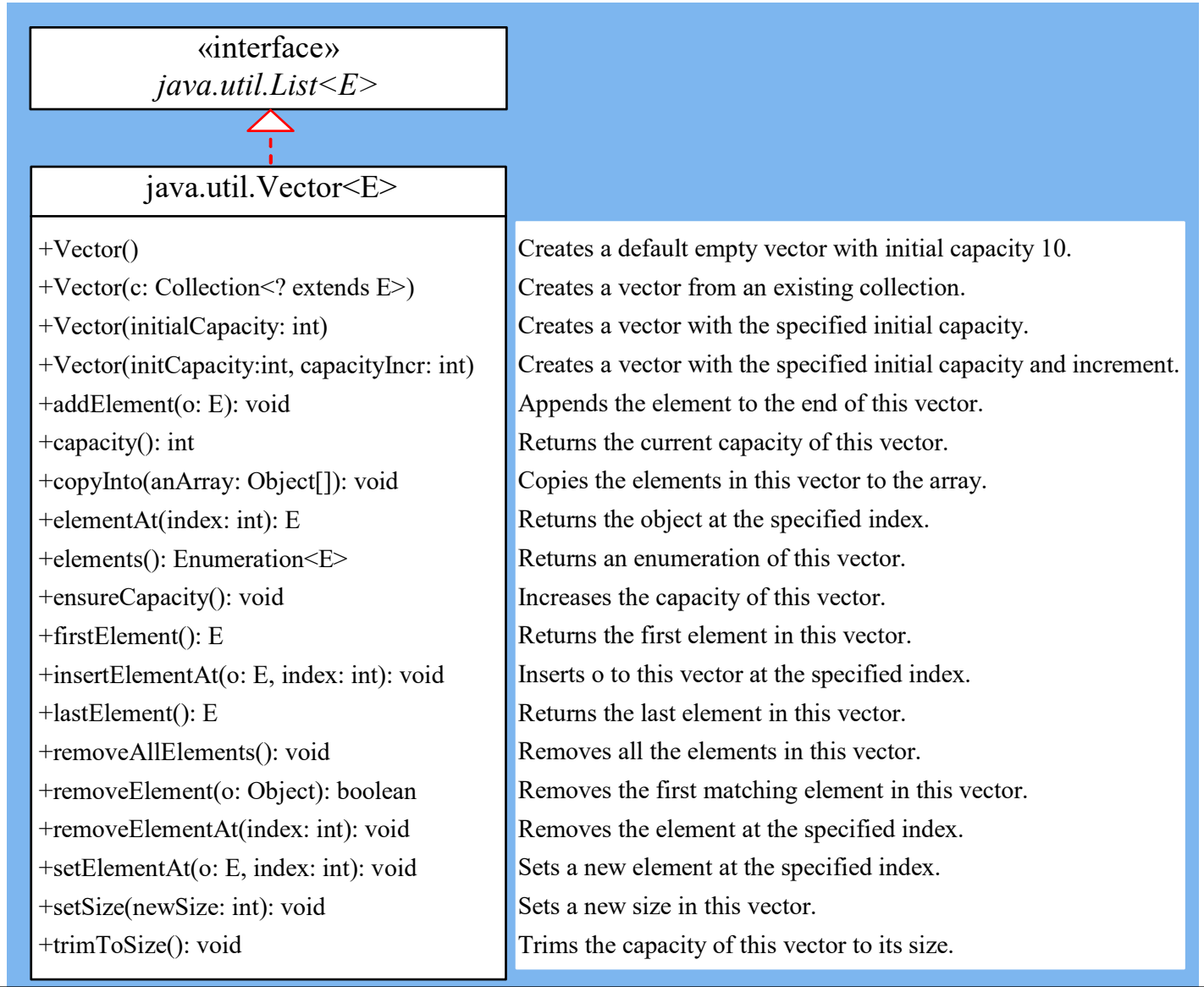
- These classes are introduced later in the section, “The Collections Class.”

# The Vector Class, cont.



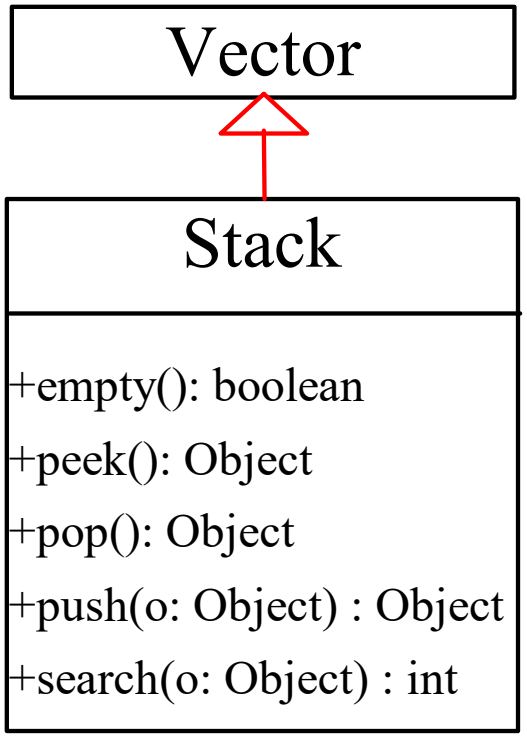

| | |
|---|---|
| +addElement(o: Object): void | Appends the element to the end of this vector |
| +capacity(): int | Returns the current capacity of this vector |
| +copyInto(anArray: Object[]): void | Copies the elements in this vector to the array |
| +elementAt(index: int): Object | Returns the object at the specified index |
| +elements(): Enumeration | Returns an emulation of this vector |
| +ensureCapacity(): void | Increases the capacity of this vector |
| +firstElement(): Object | Returns the first element in this vector |
| +insertElementAt(o: Object, index: int): void | Inserts o to this vector at the specified index |
| +lastElement(): Object | Returns the last element in this vector |
| +removeAllElements() : void | Removes all the elements in this vector |
| +removeElement(o: Object) : boolean | Removes the first matching element in this vector |
| +removeElementAt(index: int) : void | Removes the element at the specified index |
| +setElementAt(o: Object, index: int) : void | Sets a new element at the specified index |
| +setSize(newSize: int) : void | Sets a new size in this vector |
| +trimToSize() : void | Trims the capacity of this vector to its size |

# Vector class with Generics



# The Stack Class without generics

The Stack class represents a last-in-first-out stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element from the top of the stack.




Returns true if this stack is empty

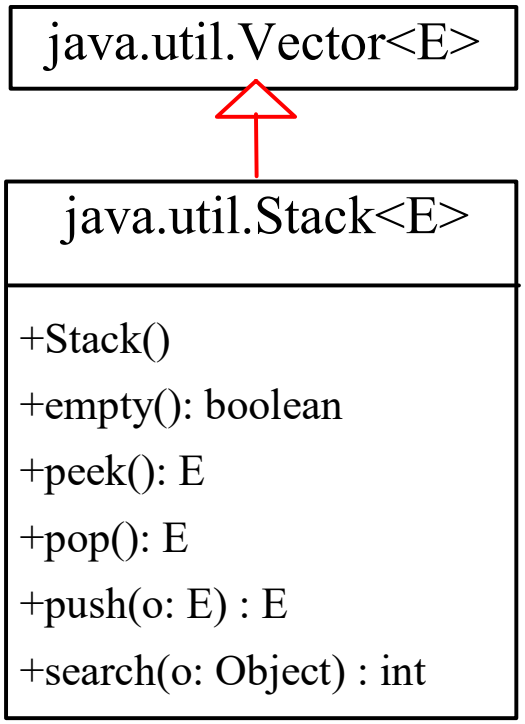
Returns the top element in this stack

Returns and removes the top element in this stack

Adds a new element to the top of this stack

Returns the position of the specified element in this stack

# Stack class with generics




Creates an empty stack.

Returns true if this stack is empty.

Returns the top element in this stack.

Returns and removes the top element in this stack.

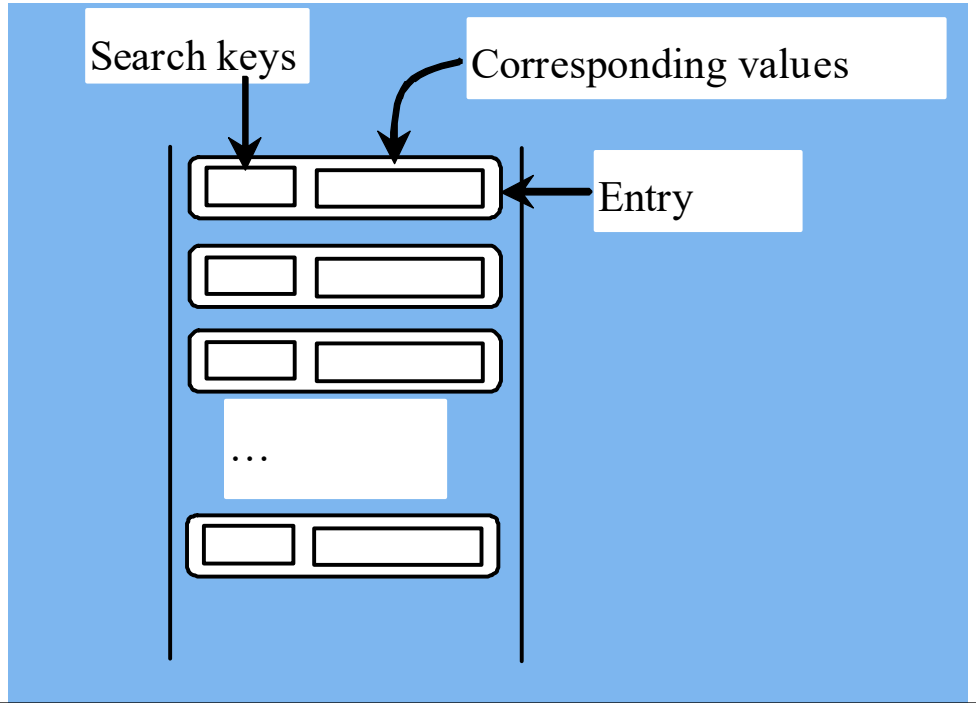
Adds a new element to the top of this stack.

Returns the position of the specified element in this stack.

# The Map Interface

The Map interface maps keys to the elements.

- Keys are like indexes.
- In List, the indexes are integer.
- In Map, the keys can be any objects.



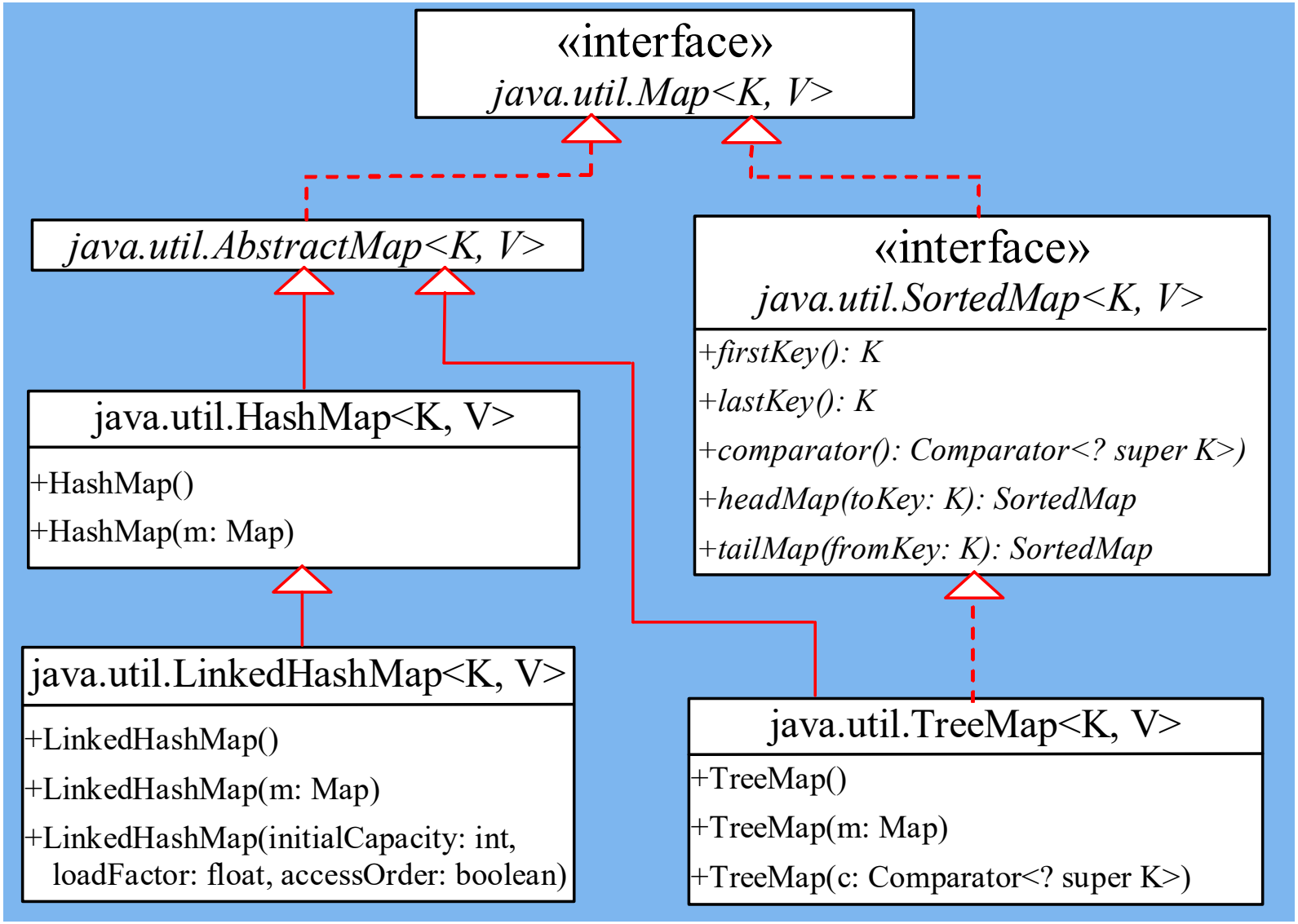
# The Map Interface UML Diagram With Generics

| *java.util.Map<K, V>* | |
|---|---|
| *+clear(): void* | Removes all mappings from this map. |
| *+containsKey(key: Object): boolean* | Returns true if this map contains a mapping for the specified key. |
| *+containsValue(value: Object): boolean* | Returns true if this map maps one or more keys to the specified value. |
| *+entrySet(): Set* | Returns a set consisting of the entries in this map. |
| *+get(key: Object): V* | Returns the value for the specified key in this map. |
| *+isEmpty(): boolean* | Returns true if this map contains no mappings. |
| *+keySet(): Set<K>* | Returns a set consisting of the keys in this map. |
| *+put(key: K, value: V): V* | Puts a mapping in this map. |
| *+putAll(m: Map): void* | Adds all the mappings from m to this map. |
| *+remove(key: Object): V* | Removes the mapping for the specified key. |
| *+size(): int* | Returns the number of mappings in this map. |
| *+values(): Collection<V>* | Returns a collection consisting of the values in this map. |

# The Map Interface UML Diagram Without Generics

| *Map* | |
|---|---|
| *+clear(): void* | Removes all mappings from this map |
| *+containsKey(key: Object): boolean* | Returns true if this map contains a mapping for the specified key. |
| *+containsValue(value: Object): boolean* | Returns true if this map maps one or more keys to the specified value. |
| *+entrySet(): Set* | Returns a set consisting of the entries in this map |
| *+get(key: Object): Object* | Returns the value for the specified key in this map |
| *+isEmpty(): boolean* | Returns true if this map contains no mappings |
| *+keySet(): Set* | Returns a set consisting of the keys in this map |
| *+put(key: Object, value: Object): Object* | Puts a mapping in this map |
| *+putAll(m: Map): void* | Adds all mappings from m to this map |
| *+remove(key: Object): Object* | Removes the mapping for the specified key |
| *+size(): int* | Returns the number of mappings in this map |
| *+values(): Collection* | Returns a collection consisting of values in this map |

# Concrete Map classes



# **`HashMap`** and **`TreeMap`**

2 concrete implementations of the Map interface: HashMap and TreeMap classes

HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping.

TreeMap class, implementing SortedMap, is efficient for traversing the keys in a sorted order.

# LinkedHashMap

LinkedHashMap

- introduced in JDK 1.4.
- It extends HashMap with a linked list implementation that supports an ordering of the entries in the map.

Entries in a HashMap are not ordered

Entries in a LinkedHashMap can be retrieved in some order

# LinkedHashMap

Two orders to retrieve entries in a LinkedHashMap

- the insertion order

- or (access order) from least recently accessed to most recently

-How?

The no-arg constructor constructs a LinkedHashMap with the insertion order.

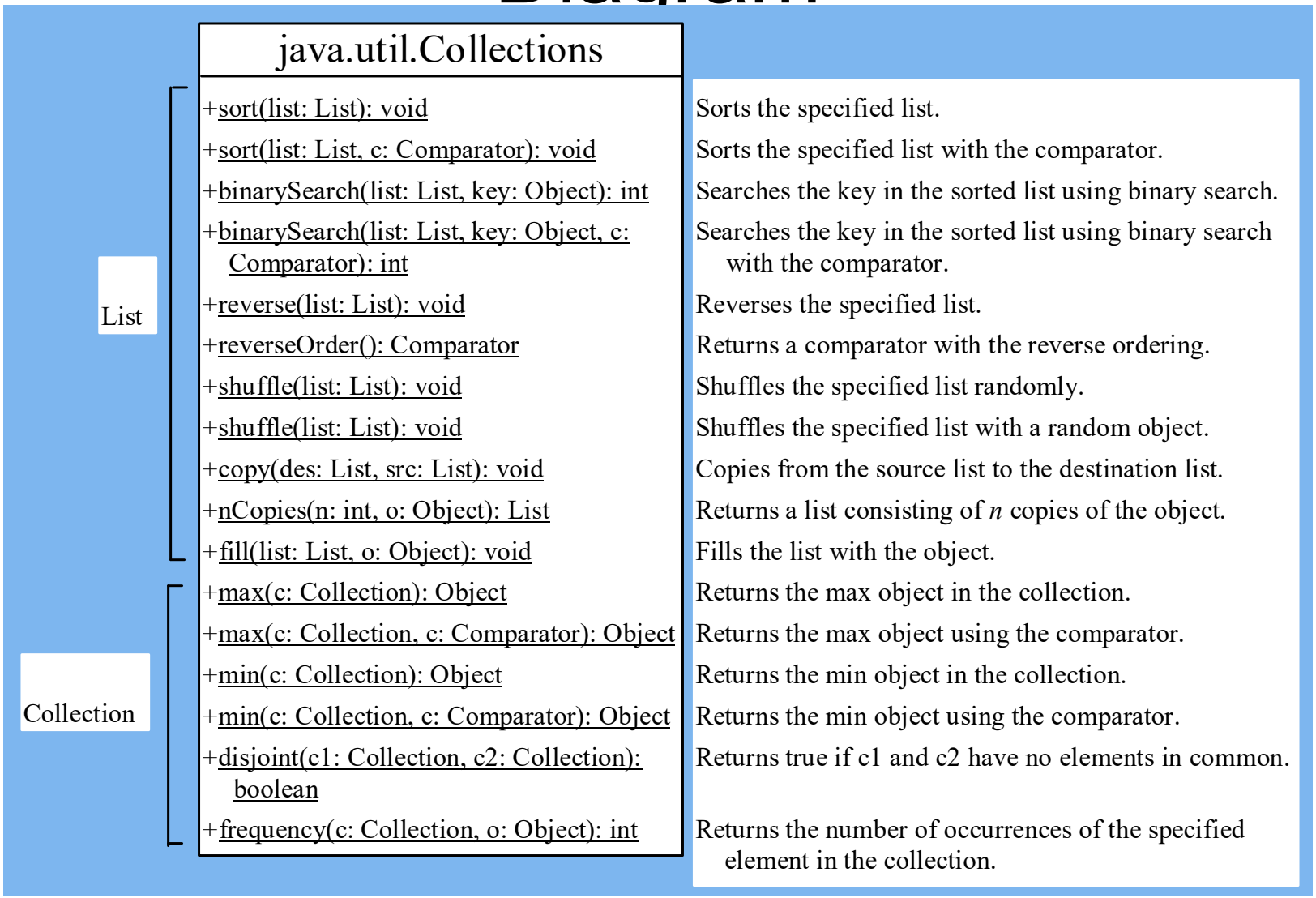
To construct a LinkedHashMap with the access order, use the LinkedHashMap(initialCapacity, loadFactor, true).

# The Collections Class

The Collections class contains

- various static methods for operating on collections
- maps, for creating synchronized collection classes, and
- for creating read-only collection classes.

# The Collections Class UML Diagram



# List Algorithms

- Most polymorphic algorithms in the **Collections** class apply specifically to List.
  - **sort** — sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A stable sort is one that does not reorder equal elements.)
  - **shuffle** — randomly permutes the elements in a List.
  - **reverse** — reverses the order of the elements in a List.
  - **rotate** — rotates all the elements in a List by a specified distance.
  - **swap** — swaps the elements at specified positions in a List.

# List Algorithms

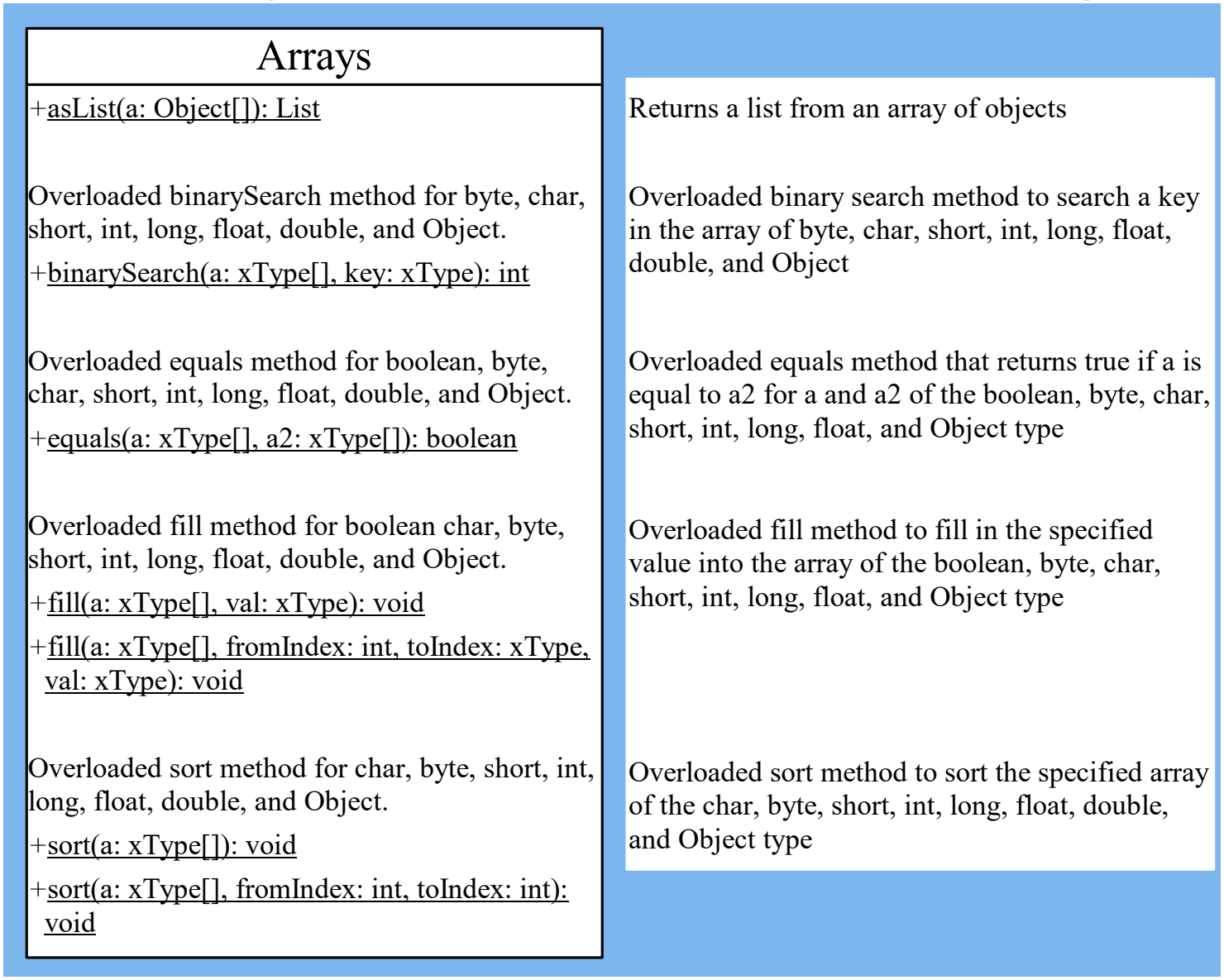
- **replaceAll** — replaces all occurrences of one specified value with another.
- **fill** — overwrites every element in a List with the specified value.
- **copy** — copies the source List into the destination List.
- **binarySearch** — searches for an element in an ordered List using the binary search 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.

# The Arrays Class

The Arrays class contains:

- various static methods for sorting
- searching arrays,
- for comparing arrays,
- and for filling array elements.
- It also contains a method for converting an array to a list.

# The Arrays Class UML Diagram



# Summary for today

- Collection
  - Set
    - HashSet
      - LinkedHashSet
    - TreeSet
  - List
    - ArrayList
    - LinkedList
    - Vector
      - Stack
- Map
  - HashMap
    - LinkedHashMap
  - TreeMap
- Collections

# References


- http://math.hws.edu/javanotes/
- http://docs.oracle.com/javase/tutorial/collections/
- http://www2.mta.ac.il/~amirk/java/presentations/03-Collections.ppt
- http://www.uwosh.edu/faculty_staff/huen/262/f09/slides/20_ch22_Collections.ppt
- https://examples.javacodegeeks.com/core-java/util/comparator/java-comparator-example/
- https://www3.ntu.edu.sg/home/ehchua/programming/java/J5c_Collection.html