- **Java Collections**

Choosing the best data structures and algorithms for a particular task is one of the keys to developing high-performance software. A data structure is a collection of data organized in some fashion. The structure not only holds data, but also supports operations for accessing and manipulating the data. Without knowing data structures, you can still write programs, but your program may not be efficient. With a good knowledge of data structures, you can build efficient programs, which are important for practical applications.

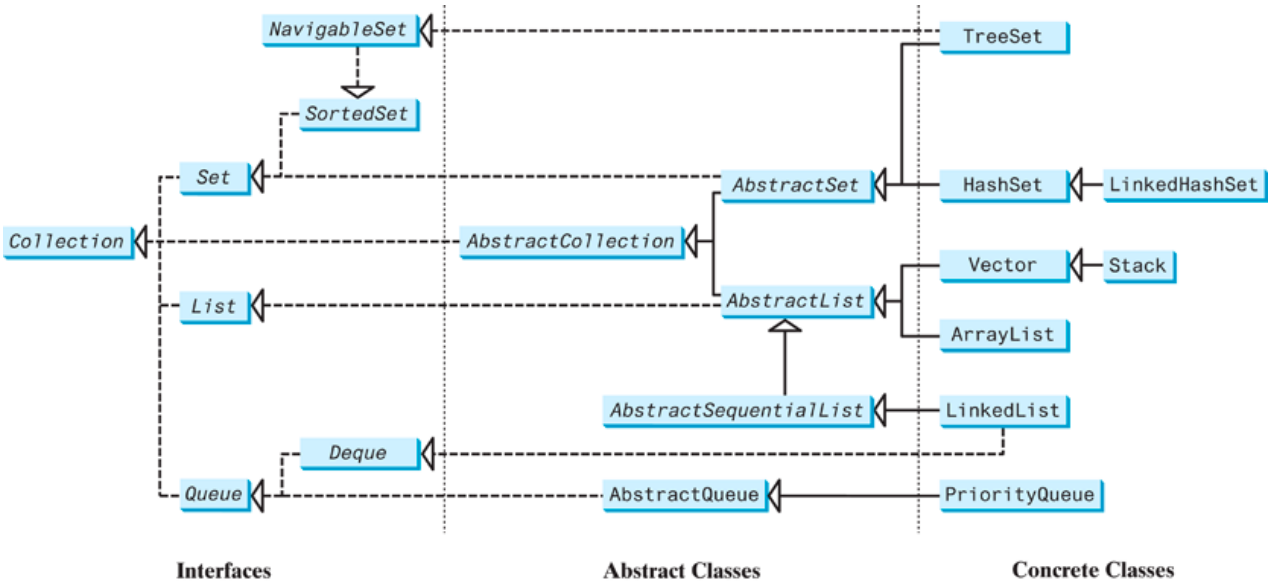
In object-oriented thinking, a data structure, also known as a **container** or **container object**, is an object that holds other objects, referred to as data or elements. For example, Poker hand (a collection of cards), Mail folder (a collection of letters), and Telephone directory (a collection of name and phone numbers).

Defining a data structure is essentially to define a class, an Abstract Data Type (ADT.) The class for a data structure should use data fields to hold data and provide methods to support such operations as **search**, **insertion**, and **deletion**. To create a data structure is therefore to create an instance from the class. You can then apply the methods on the instance to manipulate the data structure, such as inserting an element into or deleting an element from the data structure.

Java provides several data structures (lists, vectors, stacks, queues, priority queues, sets, and maps) that can be used to organize and manipulate data efficiently. These are commonly known as *Java Collections Framework*. The advantages of using the Java Collections are:

- Reduces programming effort
- Increases program quality
- Allows interoperability among unrelated APIs
- Reduces effort to learn/design new APIs
- Fosters software reuse

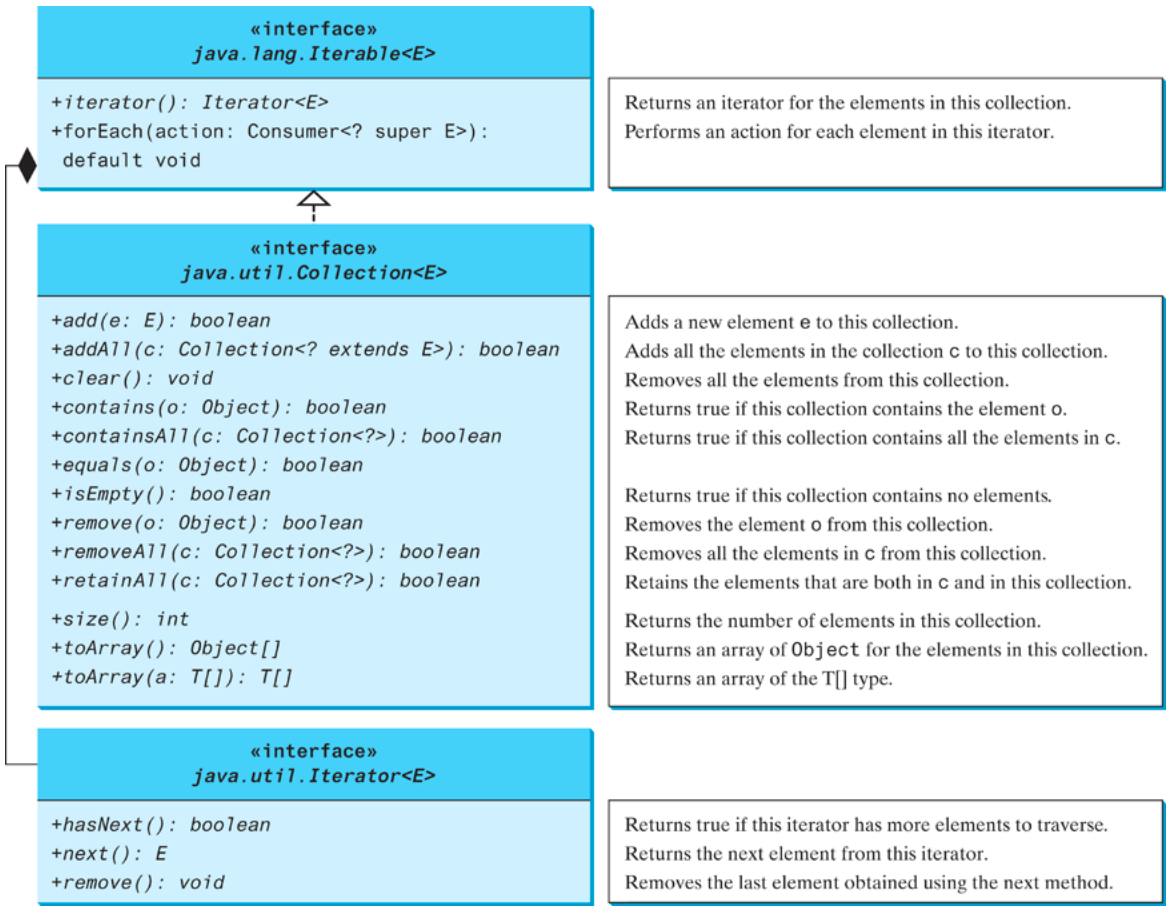
The Java **Collection interface** defines the common operations for **lists**, **vectors, stacks**, **queues**, **priority queues**, and **sets**. The Java Collections Framework supports two types of containers: one for storing a collection of elements is simply called a **collection.** The other, for storing key/value pairs, is called a **map**, which is an efficient data structures for quickly searching an element using a key.



- **Java Collections**

All the interfaces and classes defined in the Java Collections Framework are grouped in the **java.util** package. The design of the Java Collections Framework is an excellent example of using **interfaces**, **abstract classes**, and **concrete classes**. The interfaces define the common operations. The abstract classes provide partial implementation. The concrete classes implement the interfaces with concrete data structures. Providing an abstract class that partially implements an interface makes it convenient for the user to write the code. The user can simply define a concrete class that extends the abstract class rather than implementing all the methods in the interface. The abstract classes such as **AbstractCollection** are provided for convenience. For this reason, they are called **convenience abstract classes**.

The Collection interface is the root interface for manipulating a collection of objects. Its public methods are listed below.
(https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collection.html)



The Collection interface contains the methods for manipulating the elements in a collection, and you can obtain an **iterator object** for traversing elements in the collection. The Collection interface provides the basic operations for adding and removing elements in a collection. The **add** method adds an element to the collection. The **addAll** method adds all the elements in the specified collection to this collection. The **remove** method removes an element from the collection. The **removeAll** method removes the elements from this collection that are present in the specified collection. The **retainAll** method retains the elements in this collection that are also present in the specified collection. All these methods return boolean. The

return value is true if the collection is changed as a result of the method execution. The **clear()** method simply removes all the elements from the collection.

The Collection interface provides various **query operations**. The **size** method returns the number of elements in the collection. The **contains** method checks whether the collection contains the specified element. The **containsAll** method checks whether the collection contains all the elements in the specified collection. The **isEmpty** method returns true if the collection is empty.

The Collection interface provides the **toArray()** method, which returns an array of Object for the collection. It also provides the **toArray(T[])** method, which returns an array of the T[] type.

All the concrete classes in the Java Collections Framework implement the **java.lang.Cloneable** and **java.io.Serializable** interfaces except that **java.util.PriorityQueue** does not implement the Cloneable interface. Thus, all instances of Collection except priority queues can be cloned and all instances of Collection can be serialized.

Commonly used containers in Java Collections:

- Set—a collection that cannot contain duplicate elements
  - Poker hand
  - A student's schedule
- List—an ordered collection (a sequence)
- Queue
  - A collection used to hold multiple elements prior to processing
  - Typically order elements in a FIFO manner
- Deque—a double ended queue that can be used both as FIFO and LIFO
- Map—an object that maps keys to values; cannot contain duplicate keys
  - HashMap, TreeMap, and Linked HashMap

List implementation in Java:

- ArrayList
  - usually the better-performing implementation
  - constant-time positional access, O(1)
  - Not synchronized (multiple threads)
- Vector
  - Same as ArrayList but synchronized (multiple threads)
  - Synchronization produces overhead
- LinkedList
  - constant-time add/remove, O(1)
  - Linear-time positional access, O(n)

- **Traversing the Collections with the Iterator**

Each collection is **Iterable**. You can obtain its Iterator object to traverse all the elements in the collection. Iterator is a classic design pattern for **walking through a data structure** without having to expose the details of how data is organized in the data structure. The Collection interface extends the Iterable interface. The Iterable interface defines the iterator method, which returns an iterator. The Iterator interface provides a uniform way for traversing elements in various types of collections. The iterator()

method in the Iterable interface returns an instance of Iterator, which provides sequential access to the elements in the collection using the **next()** method. You can also use the **hasNext()** method to check whether there are more elements in the iterator, and the **remove()** method to remove the last element returned by the iterator.

```
import java.util.*;

public class TestIterator {
    public static void main(String[] args) {
        Collection<String> collection = new ArrayList<>();
        collection.add("New York");
        collection.add("Atlanta");
        collection.add("Dallas");
        collection.add("Madison");

        Iterator<String> iterator = collection.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next().toUpperCase() + " ");
        }
        System.out.println();
    }
}
```

You can simplify the Iterator code above using a **foreach loop** without using an iterator, as follows:

```
for (String element: collection)
    System.out.print(element.toUpperCase() + " ");
```

Another example:

```
String [] colors = {“MAGENTA”, “RED”, “WHITE”, “BLUE”, “CYAN”};
List<String> list = new ArrayList<String>();
for (String color : colors)
     list.add( color );
```

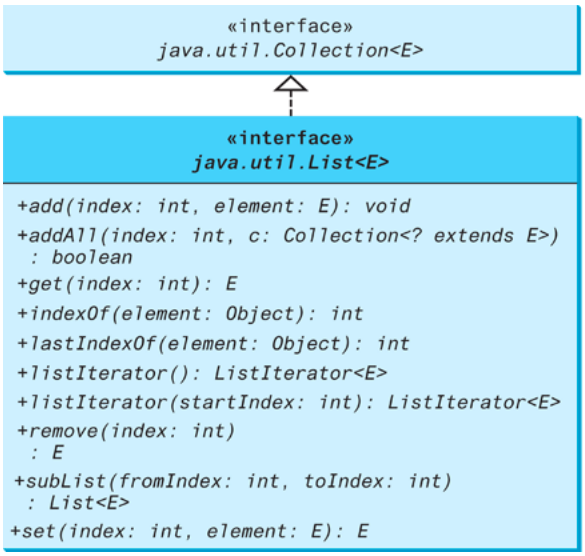
You can use the **forEach** method to perform an action for each element in a collection. The method takes an argument for specifying the action, which is an instance of a functional interface **Consumer**. The Consumer interface defines the **accept(E e) method** for performing an action on the element e. You can rewrite the preceding example using a forEach method:

```
collection.forEach(e -> System.out.print(e.toUpperCase() + " "));
```

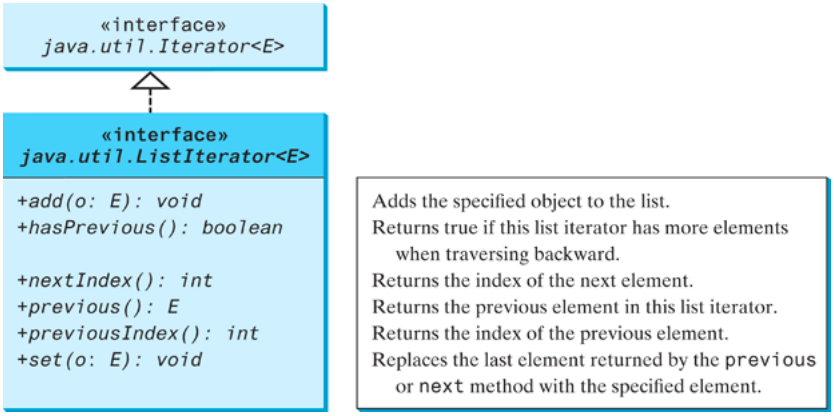
Note: If a collection is modified by one of its methods after an iterator is created, the iterator immediately becomes invalid and produce a **ConcurrentModificationExceptions.**

- **Lists**

The List interface extends the Collection interface and defines a collection for storing elements in a sequential order. To create a list, use one of its two concrete classes: **ArrayList** or **LinkedList**. The List interface extends Collection to define an ordered collection with duplicates allowed. It adds position-oriented operations as well as a new list iterator that enables a list to be traversed bidirectionally.



The add(index, element) method is used to insert an element at a specified index and the addAll(index, collection) method to insert a collection of elements at a specified index. The remove(index) method is used to remove an element at the specified index from the list. A new element can be set at the specified index using the set(index, element) method. The indexOf(element) method is used to obtain the index of the specified element's first occurrence in the list and the lastIndexOf(element) method to obtain the index of its last occurrence. A sublist can be obtained by using the subList(fromIndex, toIndex) method. The listIterator() or listIterator(startIndex) method returns an instance of ListIterator. The **ListIterator** interface extends the Iterator interface to add **bidirectional traversal** of the list.



The add(element) method inserts the specified element into the list. The element is inserted immediately before the next element that would be returned by the **next()** method defined in the Iterator interface, if any, and after the element that would be returned by the **previous()** method, if any. If the list doesn't contain any elements, the new element becomes the sole element in the list. The AbstractSequentialList class extends AbstractList to provide support for linked lists.

The **ArrayList** (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html) class and the **LinkedList** class are two concrete implementations of the List interface. ArrayList holds elements in an array. The array is dynamically created. If the capacity of the array is exceeded, a larger new array is created and all the elements from the current array are copied to the new array. That is, its

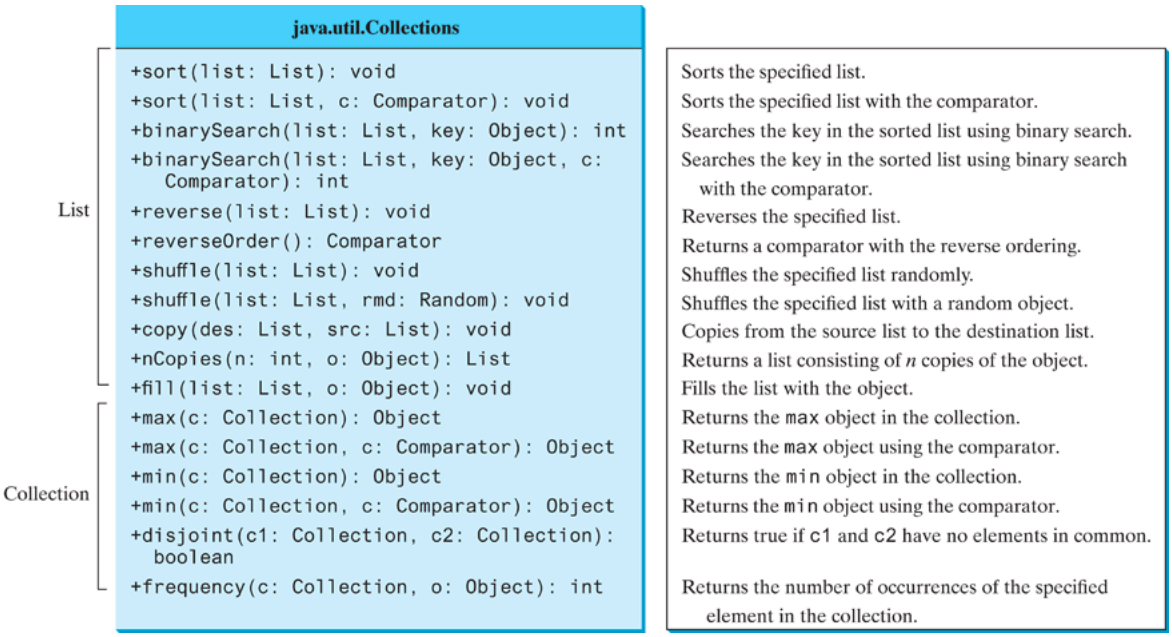
capacity **grows automatically**. An ArrayList does not automatically shrink. You can use the **trimToSize()** method to reduce the array capacity to the size of the list.
LinkedList (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/LinkedList.html)

holds elements in a linked list. Which of the two classes you use depends on your specific needs. If you need to support random access through an index without inserting or removing elements at the beginning of the list, ArrayList is the most efficient. If, however, your application requires the insertion or deletion of elements at the beginning of the list, you should choose LinkedList. A list can **grow or shrink dynamically**. Once it is created, an array is fixed. If your application does not require the insertion or deletion of elements, an array is the most efficient data structure.

- **Static Methods for Lists and Collections**

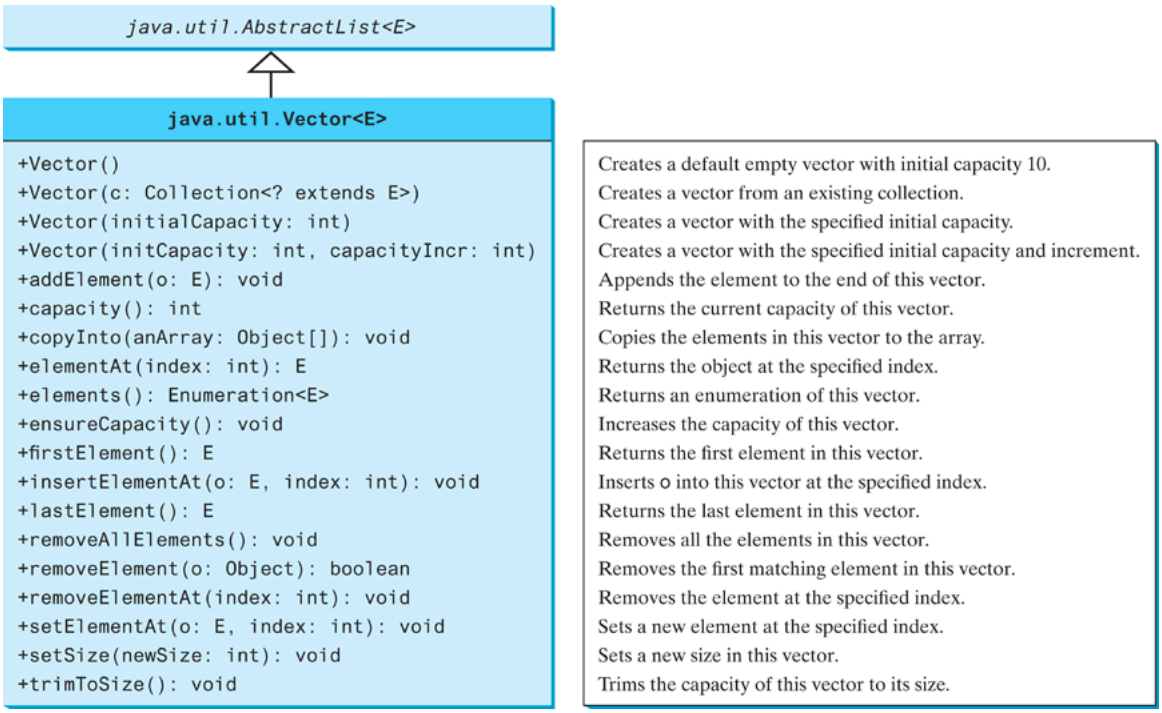
The Java class **Collections** provides some static methods that provides common algorithms to manipulate the collections:
(https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html)

  - Sort—using a merge sort algorithm, a fast and stable sort; for example, **Collections.sort(list)**
  - Shuffle—destroying any trace of order, assuming a fair source of randomness, for example, implementing games of chance
  - Reverse—reverse the order; Binary Search; copy, swap, fill, …



- **Vector and Stack classes**

Vector is a subclass of AbstractList and **Stack** is a subclass of Vector in the Java API. Vector is the same as ArrayList, except that it contains **synchronized methods** for accessing and modifying the vector. Synchronized methods can prevent data corruption when a vector is accessed and modified by two or more threads concurrently. For the many applications that do not require synchronization, using ArrayList is more efficient than using Vector.



In the Java Collections Framework, **Stack** is implemented as an extension of Vector. The empty() method is the same as isEmpty(). The peek() method looks at the element at the top of the stack without removing it. The pop() method removes the top element from the stack and returns it. The push(Object element) method adds the specified element to the stack. The search(Object element) method checks whether the specified element is in the stack.