Software Construction & Development

WEEK 04

Collections

The Collections Framework

The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects.

The Collections Framework was designed to be **high performance**. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.

The Collections Framework

The entire Collections Framework is built upon a set of standard interfaces.

Algorithms are another important part of the collection mechanism.

- Algorithms operate on collections and are defined as static methods within the Collections class.

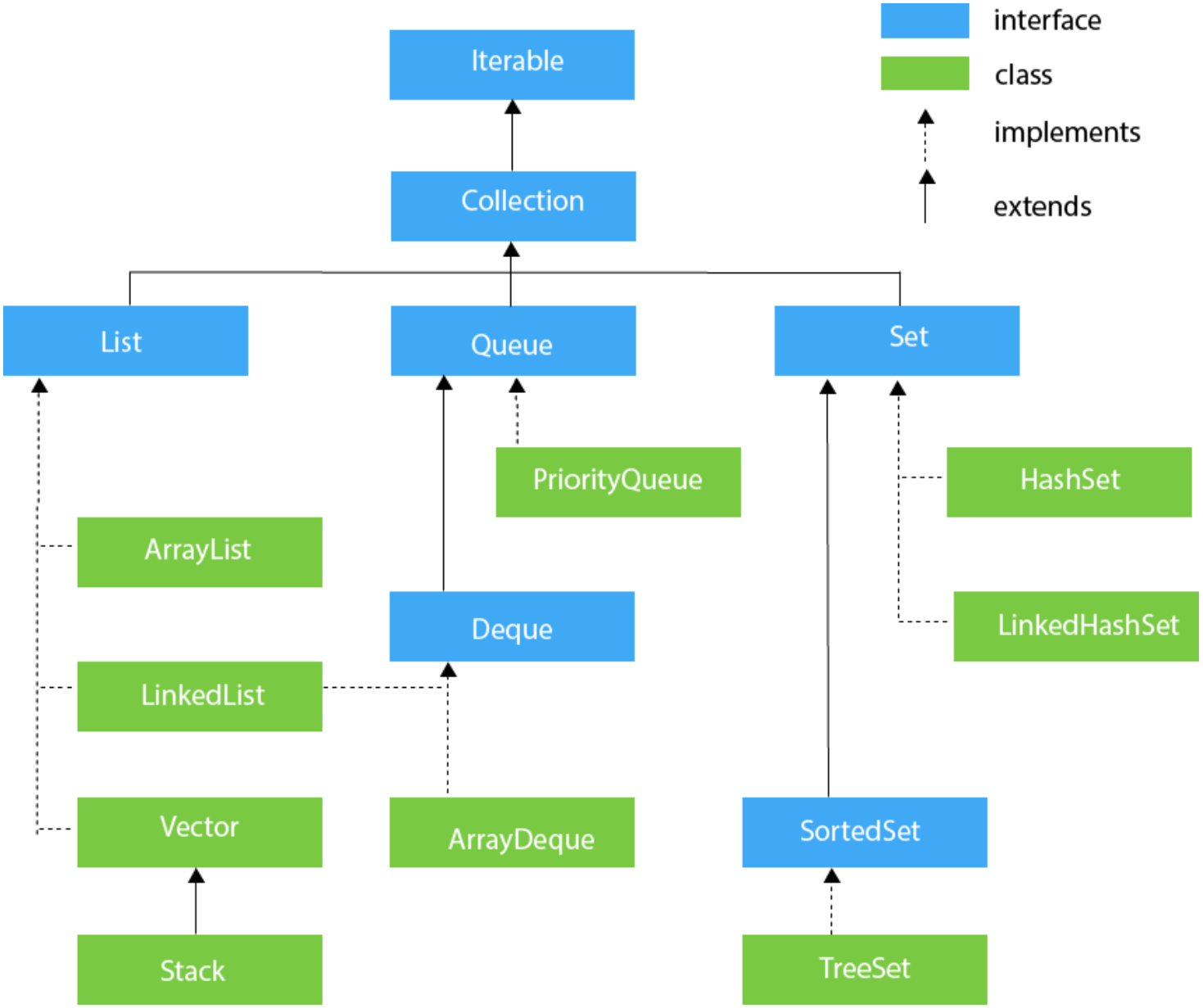
The Collections Framework

Iterator interface is closely associated with the Collections Framework.

- An iterator offers a general-purpose, standardized way of accessing the elements within a collection, one at a time.

An iterator provides a means of enumerating the contents of a collection

Spliterators are iterators that provide support for parallel iteration


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

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. Collection is a generic interface that has this declaration:

```
interface Collection<E>
```

The List Interface

The List interface extends Collection and 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. List is a generic interface that has this declaration:

```
interface List <E>
```

The Set Interface

The Set interface defines a set. It extends Collection and specifies the behavior of a collection that does not allow duplicate elements. Therefore, 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. Set is a generic interface that has this declaration:

```
interface Set <E>
```

The SortedSet Interface

The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order. SortedSet is a generic interface that has this declaration:

```
interface SortedSet <E>
```

The Queue Interface

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. Queue is a generic interface that has this declaration:

```
interface Queue <E>
```

The Deque Interface

The Deque interface extends Queue and declares the behavior of a double-ended queue. Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks. Deque is a generic interface that has this declaration:

```
interface Deque <E>
```

Self Study

TABLE 19-1, 19-2, 19-3, 19-5, 19-6 FROM BOOK (JAVA THE COMPLETE REFERENCE 11TH ED)

Table 19-1 The Methods Declared by **Collection**
Table 19-2 The Methods Declared by **List**
Table 19-3 The Methods Declared by **SortedSet**
Table 19-4 The Methods Declared by **NavigableSet**
Table 19-5 The Methods Declared by **Queue**
Table 19-6 The Methods Declared by **Deque**

The Collection Classes

Collection Classes implement Collection Interfaces

- Some of the classes provide full implementations that can be used as-is.
- Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections.

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**.

Collection Classes (that we will discuss)

- ArrayList
- LinkedList
- HashSet

Iterator/ListIterator

- **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.

- Both are generic interfaces:
  - `interface Iterator <E>`
  - `interface ListIterator <E>`

Iterator Methods

Method	Description
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection.
boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
E next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
default void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**. The default version throws an **UnsupportedOperationException**.

Table 19-8 The Methods Declared by **Iterator**

ListIterator Methods

Method	Description
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection.
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

Table 19-9 The Methods Provided by **ListIterator**

Using an Iterator

To use an iterator to cycle through the contents of a collection, follow these steps:

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( )`.

Using a ListIterator

For collections that implement List, you can obtain an iterator by calling `listIterator( )`.

NOTE: ListIterator is available **only** to those collections that implement the List interface.

```
class IteratorDemo {
  public static void main(String args[]) {
    // Create an array list.
    ArrayList<String> al = new ArrayList<String>();

    // Add elements to the array list.
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");

    // Use iterator to display contents of al.
    System.out.print("Original contents of al: ");
    Iterator<String> itr = al.iterator();
    while(itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // Modify objects being iterated.
    ListIterator<String> litr = al.listIterator();
    while(litr.hasNext()) {
      String element = litr.next();
      litr.set(element + "+");
    }
```

```
    System.out.print("Modified contents of al: ");
    itr = al.iterator();
    while(itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // Now, display the list backwards.
    System.out.print("Modified list backwards: ");
    while(litr.hasPrevious()) {
      String element = litr.previous();
      System.out.print(element + " ");
    }
    System.out.println();
  }
}
```

Output:

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

Using For-Each

`for` can cycle through any collection of objects that implement the Iterable interface

Useful if:

- You won't be modifying the contents of a collection
- You won't be obtaining elements in reverse order

```
class ForEachDemo {
  public static void main(String args[]) {
    // Create an array list for integers.
    ArrayList<Integer> vals = new ArrayList<Integer>();

    // Add values to the array list.
    vals.add(1);
    vals.add(2);
    vals.add(3);
    vals.add(4);
    vals.add(5);

    // Use for loop to display the values.
    System.out.print("Contents of vals: ");
    for(int v : vals)
      System.out.print(v + " ");

    System.out.println();

    // Now, sum the values by using a for loop.
    int sum = 0;
    for(int v : vals)
      sum += v;

    System.out.println("Sum of values: " + sum);
  }
}
```

Output:

```
Contents of vals: 1 2 3 4 5
Sum of values: 15
```