Collections Framework – Part 1

Chapter 9, Core Java Volume I

Chapter 16, Java: How to Program, 10th Ed. (Deitels)

Contents

- Java Collections Framework
- Separating Collection Interfaces from Implementation
- Collection Interface
- Iterators
- Generic Utility Methods
- Interfaces in the Collection Framework
- Classes in the Collection Framework
- Concrete Collections

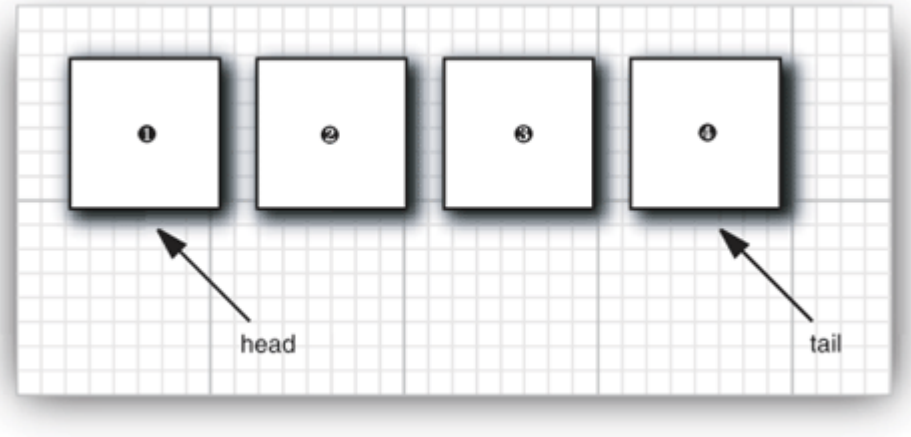
Java Collections Framework

- Java collections framework
  - a unified architecture for representing and manipulating collective objects
  - enables them to be manipulated independently of the details of their representation
  - provides interfaces and their implementations for manipulating those collections
  - reduces programming effort
- A collection is a data structure—actually, an object—that can hold references to other objects.
  - Usually, collections contain references to objects that are of the same type.
  - Collections of primitive type data
- Package `java.util`

Separating Collection Interfaces from Implementation

- Collections framework separates interfaces and implementations.
- Example: Queue (first-in, first-out)
  - Queue interface provides abstract specification
  - It tells you nothing about how the queue is implemented

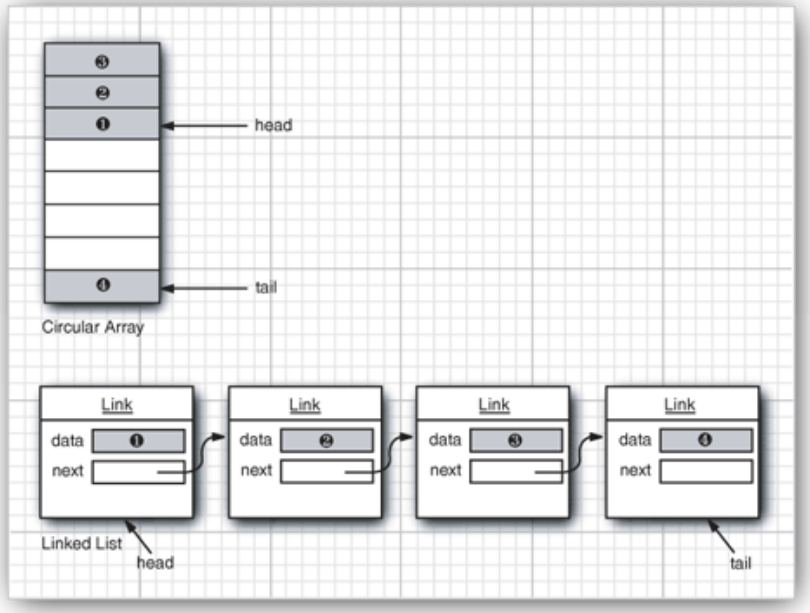
```
public interface Queue<E> // a simplified form of the interface in the standard library
{
   void add(E element);
   E remove();
   int size();
}
```



Separating Collection Interfaces from Implementation

- A collection interface can have multiple implementing classes:

```
public class CircularArrayQueue<E> implements Queue<E>
public class LinkedListQueue<E> implements Queue<E>
  // not actual library classes
```



- Use the interface to hold the collection reference
- You can easily change the implementation

```
Queue<Customer> expressLane =
        new CircularArrayQueue<>(100);
expressLane.add(new Customer("Harry");

Queue<Customer> expressLane =
        new LinkedListQueue<>();
expressLane.add(new Customer("Harry");
```

Collection Interface

- Interface Collection<E> is the root interface for collection classes in Java library.
- Interface Collection<E> contains fundamental methods such as:
  - int size()
  - boolean add(E element) // returns true if adding changes the collection
    // otherwise, returns false
  - Iterator<E> iterator()
  - …
- Interface Collection<E> provides a method that returns an Iterator<E> object, which allows a program to walk through the collection during the iteration.

Iterators

- A generic interface belongs to collection framework.
- Allows us to traverse the collection and access the data element of collection without bothering the user about specific implementation of that collection it.
- Basically a collection provides an iterator by its iterator method.
- Iterator<E> has methods:

```
public interface Iterator<E>
  E next();
  boolean hasNext();
  void remove();
  default void forEachRemaining(Consumer<? super E> action);
}
```

→

```
while (hasNext())
    action.accept(next());
```

- The types of iterator interfaces:
  - java.util.Iterator<T> provides for one-way traversal
  - java.util.ListIterator<T> provides two-way traversal

Iterators

- Typical example:

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
   String element = iter.next();     // next() method can throw a NoSuchElementExcetion
   // do something with element
}
```

- More concisely:

```
for (String element : c) // works for any Iterable
{
   do something with element
}
```

- The Collection<E> interface extends the Iterable<E> interface.

Iterators

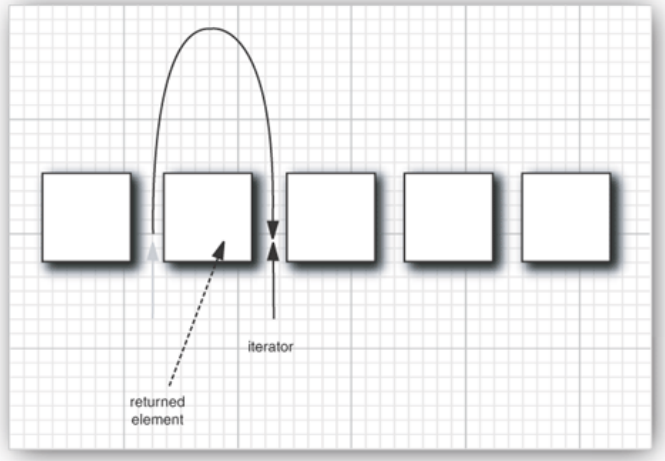
- The order in which the elements are visited depends on the collection types.
  - ArrayList : starting at index 0 and incrementing the index …
  - HashSet : in a random order
- Think of iterator position as being between elements.
- The remove method removes the element that was just returned by next:

```
Iterator<String> it = c.iterator();
it.next(); // skip over the first element
it.remove(); // now remove it
```

- Caution: Calling remove twice in a row without calling next in between is an error.

```
it.remove();
it.remove();  // error

it.remove();
it.next();
it.remove();
```



Generic Utility Methods

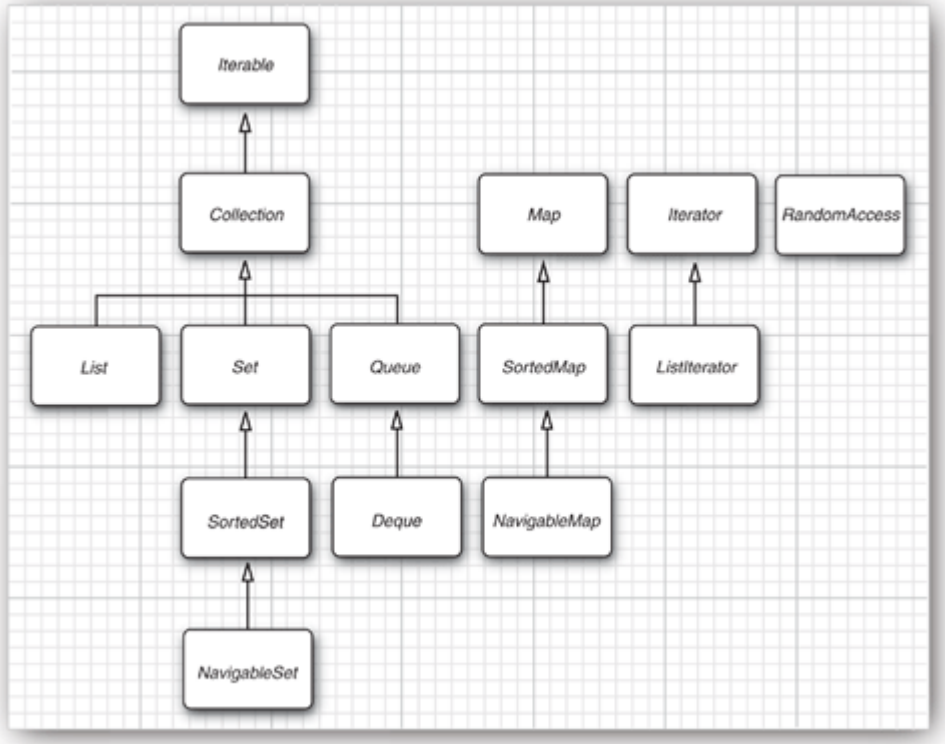
- Collection and Iterator interfaces are generic which means you can write utility methods that operate on any kind of collection.

```
public static <E> boolean contains(Collection<E> c, Object obj)
{
   for(E element : c)
     if(element.equals(obj)
          return true;
     return false
}
```

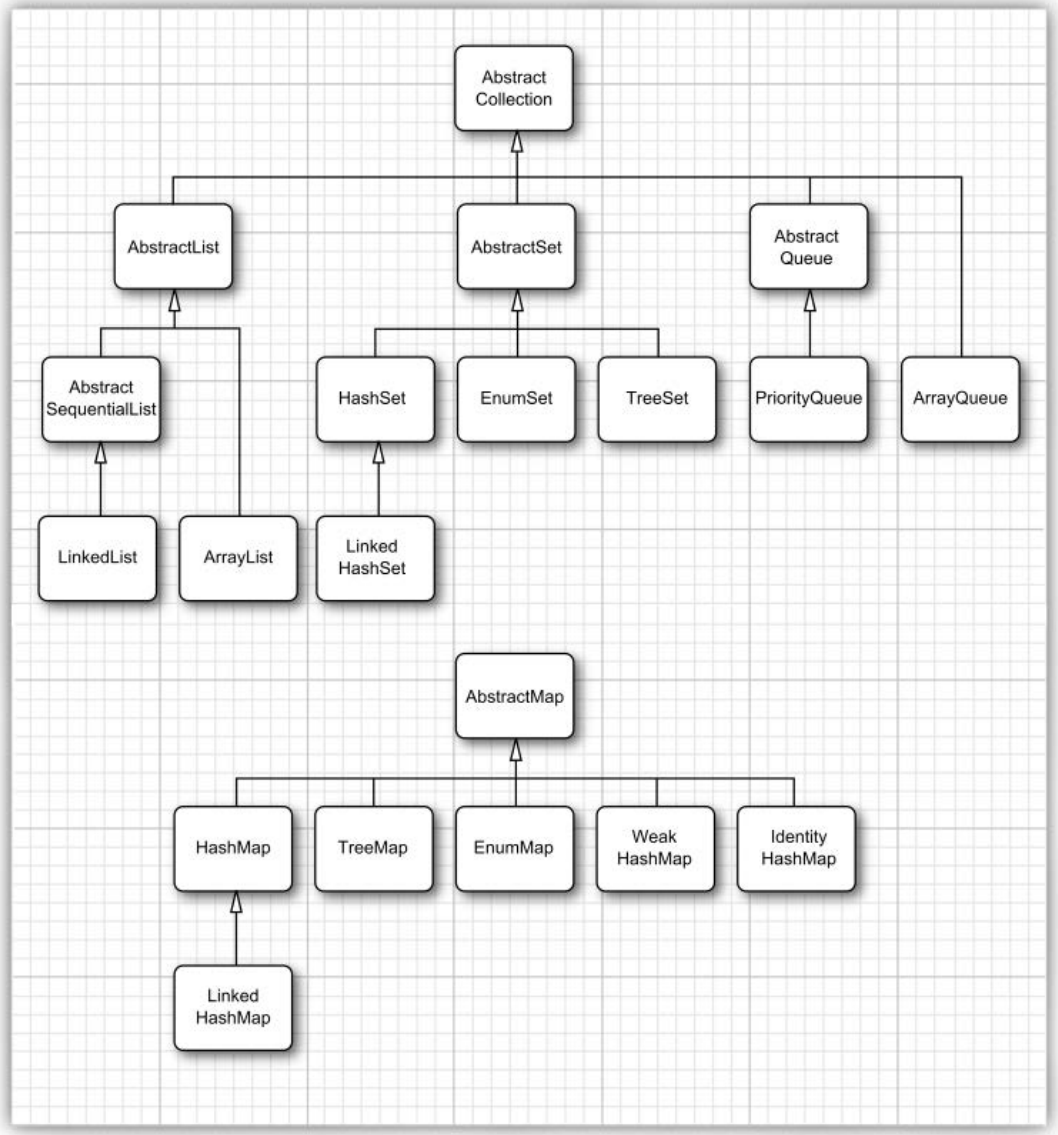
- Collection interface provides some useful methods:
  - int size()
  - boolean isEmpty()
  - boolean contains(Object obj)
  - boolean containAll(Collection <?> c)
  - boolean add(E element)
  - boolean addAll(Collection<? extends E> from)
  - boolean remove(Object obj)
  - ..

Interfaces in the Collection Framework

- **Collection** holds elements.
- **Map** holds key/value pairs.
- **List**: Ordered collection.
- **Set**: Unordered collection without duplicates.
- **SortedSet/SortedMap**: Traversed in sorted order.
- **NavigableSet/NavigableMap**: Additional methods for sorted sets/maps.
  - extends **SortedSet/SortedMap** interfaces.
- Why doesn't **Map** extend **Collection**?
  - A **Map** is not a set of pair<key, value>.



Classes in the Collection Framework



Concrete Collections

- `ArrayList` - An indexed sequence that grows and shrinks dynamically
- `LinkedList` - An ordered sequence that allows efficient insertion and removal at any location
- `ArrayDeque` - A double-ended queue that is implemented as a circular array
- `HashSet` - An unordered collection that rejects duplicates
- `TreeSet` - A sorted set
- `EnumSet` - A set of enumerated type values
- `LinkedHashSet` - A set that remembers the order in which elements were inserted
- `PriorityQueue` - A collection that allows efficient removal of the smallest element
- `HashMap` - A data structure that stores key/value associations
- `TreeMap` - A map in which the keys are sorted
- `EnumMap` - A map in which the keys belong to an enumerated type
- …