Collections Framework - Part 2

Chapter 9, Core Java Volume I

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

Contents

- Lists
- Example: ArrayList and Interator
- Linked Lists
- Concurrent Modifications
- Example: LinkedList
- Implementing Iterator Classes

Lists

- Linear traversal with iterators
- Random access with integer indexes.
- `List<E>` methods:

```
void add(int index, E element)
void remove(int index)
E get(int index)
E set(int index, E element)
```

- `ListIterator<E>` extends `Iterator<E>` and provides methods:

```
void add(E element)   // add an element before iterator position
E previous()
boolean hasPrevious();
void set(E element)   // replace the last element by next() or previous()
```

- `ArrayList` and `LinkedList` implement `List` interface

Lists

- List Operations and Iterators

```
List<String> staff = new ArryList<>();
staff.add("Amy");  // add method adds an element at the end of the list
staff.add("Bob");
staff.add("Carl");
Iterator<String> iter = staff.iterator();
String first = iter.next();
String second = iter.next();
iter.remove();      // remove last visited element (i.e. second one)
```

|ABC - initial
A|BC - after first iter.next()
AB|C - after second iter.next()
A|C - after iter.remove()

Example: ArrayList and Iterator (Deitel's Fig. 16.2)

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
public class CollectionTest
{
   public static void main(String[] args)
   {
      String[] colors = {"MAGENTA", "RED", "WHITE", "BLUE",
"CYAN"};
      List<String> list = new ArrayList<String>();

      for (String color : colors)
         list.add(color); // adds color to end of list

      String[] removeColors = {"RED", "WHITE", "BLUE"};
      List<String> removeList = new ArrayList<String>();
      for (String color : removeColors)
         removeList.add(color);
      // output list contents
      System.out.println("ArrayList: ");

      for (int count = 0; count < list.size(); count++)
         System.out.printf("%s ", list.get(count));

      // remove from list the colors contained in removeList
      removeColors(list, removeList);

      // output list contents
      System.out.printf
         ("%n%nArrayList after calling removeColors:%n");

      for (String color : list)
         System.out.printf("%s ", color);
   }
```

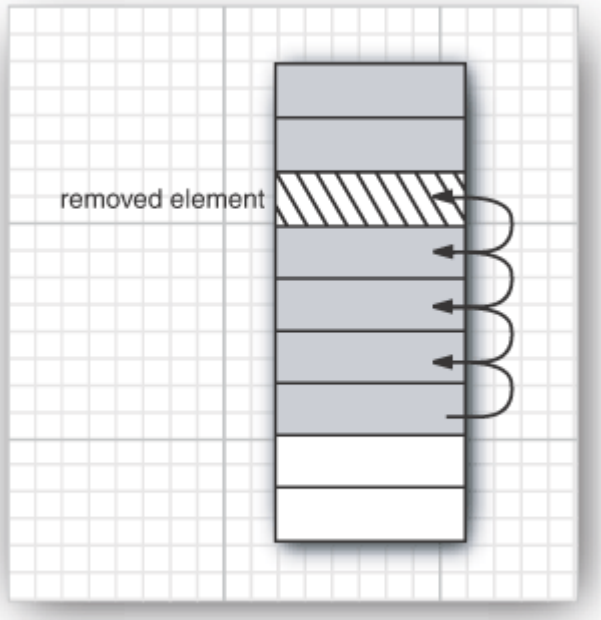
Example: ArrayList and Iterator (Deitel's Fig. 16.2)

```
// remove colors specified in collection2 from collection1
 private static void removeColors(Collection<String> collection1, Collection<String> collection2)
 {
    // get iterator
    Iterator<String> iterator = collection1.iterator();

    // loop while collection has items
    while ( iterator.hasNext() )
    {
       if (collection2.contains( iterator.next()))
          iterator.remove(); // remove current element
    }
 }
} // end class CollectionTest
```

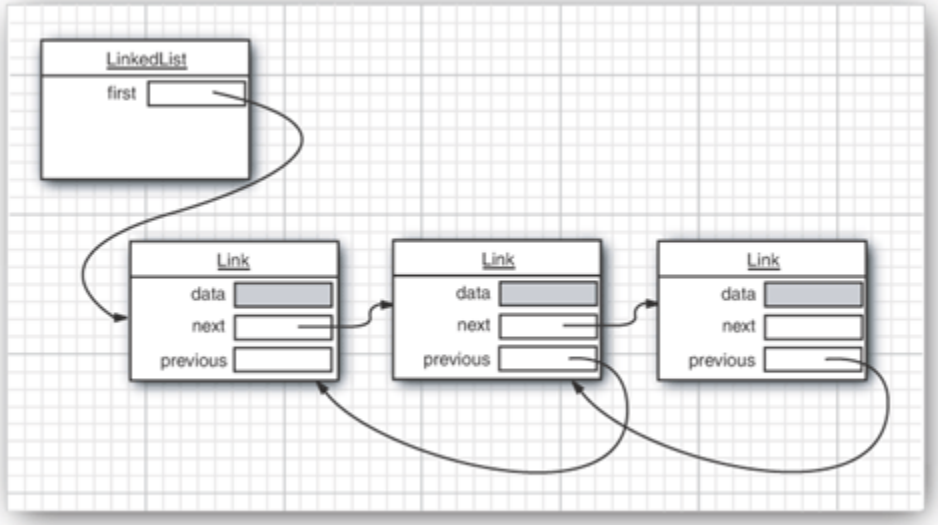
Linked Lists

- Array lists manage an array that can grow or shrink.
- What's not to like?
  - Inserting and removing in the middle is slow:

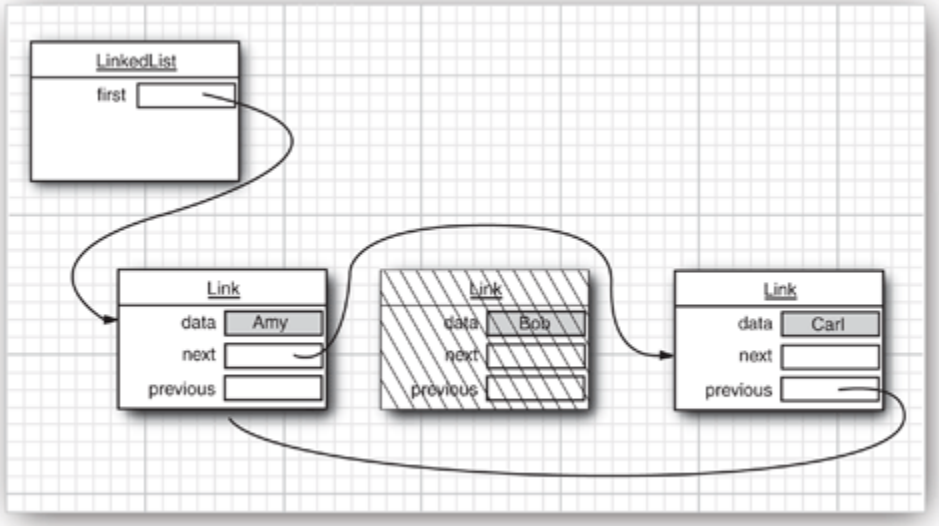


Linked Lists

- Linked list=chain of “links”:



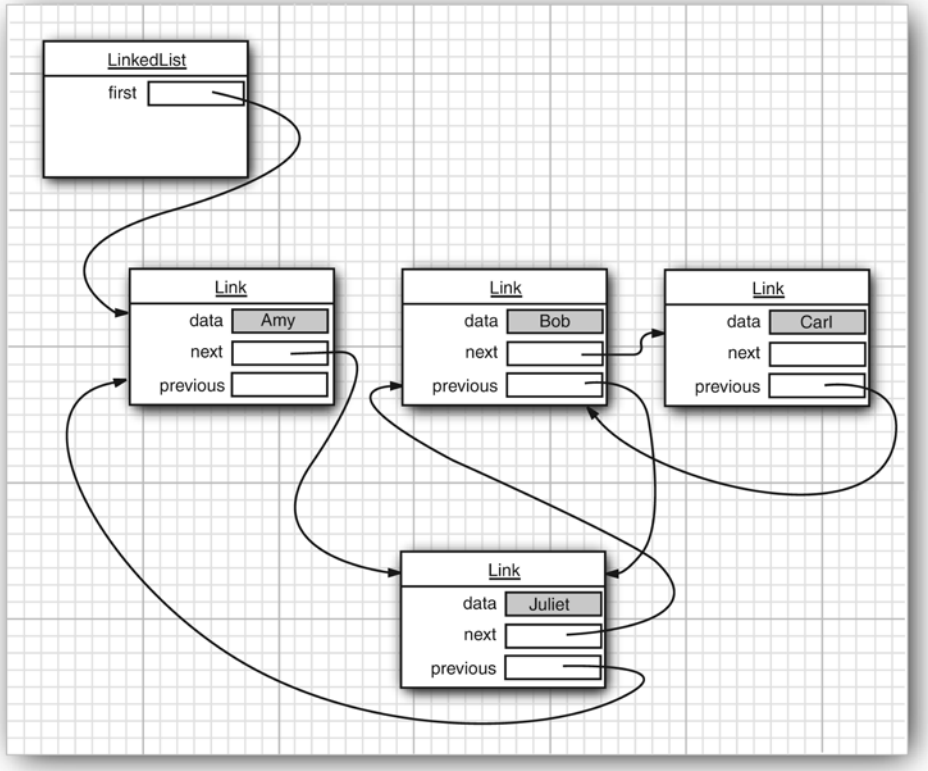
- Easy to remove in the middle:



Linked Lists

- ListIterator Operations

```
List<String> staff = new LinkedList<>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.ListIterator();
iter.next(); // skip the first element
iter.add("Juliet");
    // adds "Juliet" before the second element
```



- If you add an element with a new list iterator, it is added to the beginning of the list.
- When you can add multiple times, the elements are simply added in turn before the current iterator positions.
- If you call previous method and then remove method, the skipped (right) element is removed.

Linked Lists

- A set method replaces the last element, returned by a call to next or previous, with a new element

```
List<String> staff = new LinkedList<>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.ListIterator();
String oldValue = iter.next(); // returns the first element  "Amy"          A|BC
iter.set("Juliet");            // replace the first element with "Juliet"   J|BC
```

- Iterating a list in reverse order

```
ListIterator<String> iter = staff.ListIterator(staff.size());    // ABC|
while(iter.hasPrevious())
{
    System.out.println(iter.previous());
}
```

Concurrent Modifications

- Suppose one iterator traverses a collection.
- At the same time, another modifies the collection by removing or adding elements.
  - through the underlying collection
  - through another iterator
- In the case of a linked list, that won't work—the links will not be consistent.
- Linked list iterators detect concurrent modifications:

```
List<String> list = . . .;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next();      // throws ConcurrentModificationException
```

- To avoid the concurrent modifications:
  - Ok to have multiple readers and no writer.
  - Ok to have single iterator that can both read and write

Example: LinkedList (Listing 9.1)

```
public class LinkedListTest
{
   public static void main(String[] args)
   {
      List<String> a = new LinkedList<>();
      a.add("Amy");
      a.add("Carl");
      a.add("Erica");

      List<String> b = new LinkedList<>();
      b.add("Bob");
      b.add("Doug");
      b.add("Frances");
      b.add("Gloria");

      // merge the words from b into a
      ListIterator<String> aIter = a.listIterator();
      Iterator<String> bIter = b.iterator();
      while (bIter.hasNext())
      {
         if (aIter.hasNext()) aIter.next();
         aIter.add(bIter.next());
      }
      System.out.println(a);
      // remove every second word from b
      bIter = b.iterator();
      while (bIter.hasNext())
      {
         bIter.next(); // skip one element
         if (bIter.hasNext())
         {
            bIter.next(); // skip next element
            bIter.remove(); // remove that element
         }
      }
```

Example: LinkedList (Listing 9.1)

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

    // bulk operation: remove all words in b from a

    a.removeAll(b);

    System.out.println(a);
  }
}
```

```
initial data
   a = A C E
   b = B D F G
after merging
   a = A B C D E F G
   b = B D F G
after removing every second element from b
   b = B F
after remove all words in b from a
   a = A C D E G
   b = B F
```

Implementing Iterator Classes

- Each concrete class has its own iterator classes