Collection Types: The List Interface

- A List is an ordered sequence of elements: $a_1, a_2, a_3, \ldots, a_n$.
- It is based on an array, that grows as needed to hold the elements.
- By using List as the type of the variable, it’s easy to switch implementations when needed.

ArrayList and LinkedList implement List

- Just as HashMap and TreeMap implement Map, and HashSet and TreeSet implement Set, we have two major implementations of the List interface, ArrayList and LinkedList.
- ArrayList is based on an array, that grows as needed to hold the elements of the List.
- LinkedList is the expected chain-listed implementation.
- Good news: we don’t need equals/hashCode/compareTo for elements because the List tolerates/duplicates elements, unlike Set.
- Example
  ```java
  List<BankAccount> myList = new ArrayList<BankAccount>();
  ```
  or
  ```java
  List<BankAccount> myList = new LinkedList<BankAccount>();
  ```
- By using List as the type of the variable, it’s easy to switch implementations when needed.

Collection Types: The List Iterator

- Lists have more powerful iterators than non-List Collections. We can iterate up and down the list. We can start from either end by using the pos argument of ListIterator(pos).

A Simple ListIterator Example

```java
import java.util.ArrayList;
import java.util.ListIterator;

public class TestArrayList {
    public static void main(String [] args) {
        ArrayList<String> array = new ArrayList<String>();
        array.add("apple");
        array.add("banana");
        // 1. going forward: or see enhanced for loop here
        ListIterator<String> itr = array.listIterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
        // 2. going backwards
        ListIterator<String> itr2 = array.listIterator(array.size());
        while (itr2.hasNext()) {
            System.out.println(itr2.next());
        }
        // The listIterator is still alive...
        System.out.println(itr2.next());
    }
}
```
**Collection Types: The List Interface**

- The two most important classes that implement the List interface are `LinkedList` and `ArrayList`.
- They have different performance for large lists.
- Both have extra methods over and above the List interface.
- But by far their most important methods are in the Collection plus List interfaces.
- Because of this, it is common to use List type instead of `ArrayList`.

```java
List<BankAccount> myList = new ArrayList<BankAccount>();
```

**Mental Model of a List**

- Here is a 4-member list:

```
A1 => A2 => A3 => A4
```

- We can `get(0)`, `get(3)` and access any particular object ref.
- We can set(0, b) and replace the object at 0 with b.
- What happens if we set(4, b)?
- To grow the list we need to use `add(Object x)`, but where does it go?
- This is fast because the `LinkedList` tracks the end-of-list.

**Collections in Java**

- A `ListIterator` starting from 0, has a next method that returns element 0 on first call, element 1 on second, etc.
- Should test with `hasNext` before doing a `next`.
- If `hasNext` returns false, the iterator is at end of list (EOL).
- It starts at beginning of list, so there are 5 different iterator states for 4 elements:

```
A0 > A1 > A2 > A3
```

- Original iteration, before element A0
- After first `next`, returning A1 (just before element 1)
- After 2nd `next`, returning A2
- After 3rd `next`, returning A3
- After last `next`

**Mental Model of a ListIterator**

- You can think of an iterator as sitting between two elements, so to speak.
- At each point in time, an iterator is positioned just after the element that would be returned by `previous()`, and just before the element that would be returned by `next()`.
- A ListIterator can go both ways.
- When we talk about numerical position in a list, it’s normally about the position of an element, not directly the iterator.

**Question**

- What happens if next returns A1, then another `next` returns A2, and then a previous is done – is A1 returned?
Question

- What happens if next returns A1, then another next returns A2, and then a previous is done—is A1 returned?
- No! We've just gone past A2 one way, and now we go back across it again, so A2 gets returned again.

List Client Code Example

- Here the ListIterator<Integer> goes all the way down the list to EOL, then back along the list, so the turn-around occurs at EOL.
- Another ListIterator<Integer> starts from "list.size()-1", which would be 4 for our list.
- This is an artificial element number denoting the EOL position of the iterator.
- Again, there are n+1 different iterator states for n elements, and these are numbered from 0 to n.

Recall iterator vs. remove with Sets

- With an iterator going across a Set (entrySet for a Map, end of last class), we saw that a separate Set remove invalidates the iterator, causing a ConcurrentModificationException when the iterator is used again.
- Note that this means the iterator (one object) and the set itself (another object) are in communication (via shared memory).
- Remove was made "through the iterator", i.e. by iterator's remove, and are not so disruptive the iterator is still valid.
- There are similar behaviors with iterators on Lists
- But one iterator doesn't notify another iterator when it does a remove, so we have to be careful with multiple iterators on a List...

List Client Code Example

```java
class TestArrayList4 {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<Integer>();
        int size = 4;
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);

        System.out.println("Initial: ");
        ListIterator<Integer> itr1 = list.listIterator(0);
        System.out.println((itr1.hasNext() ? "true" : "false") + " " + itr1.next()
                          + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " ").

        System.out.println("after remove(3)");
        list.remove(3);
        System.out.println((itr1.hasNext() ? "true" : "false") + " " + itr1.next()
                          + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " ").

        System.out.println("after remove(1)");
        list.remove(1);
        System.out.println((itr1.hasNext() ? "true" : "false") + " " + itr1.next()
                          + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " ").

        System.out.println("after remove(0)");
        list.remove(0);
        System.out.println((itr1.hasNext() ? "true" : "false") + " " + itr1.next()
                          + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " ").

        System.out.println("after remove(2)");
        list.remove(2);
        System.out.println((itr1.hasNext() ? "true" : "false") + " " + itr1.next()
                          + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " ").

        System.out.println("after remove(3)");
        list.remove(3);
        System.out.println((itr1.hasNext() ? "true" : "false") + " " + itr1.next()
                          + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " " + ((itr1.hasNext() ? "true" : "false") + " " + itr1.next() + " ").

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

Iterator Remove

- An iterator sits between elements.
- When calling remove, which nearby element gets removed?
- The object removed is the last one returned by next or previous, and only one remove per movement-action is allowed.
- This also holds for iterators over Sets (in fact, any Iterator)
- What happens if you next, remove, and then next again?
- You access the element just after the removed element.
- Because we've moved past the deleted element already, the iterator position is clear.
- If you next, remove, previous, you should get the previous to removed. And so on, using the model above.

Remove List Duplicates With Two Iterators?

- Example: Using two iterators to remove duplicates from a LinkedList doesn't work!
- Proposed Algorithm:
  - Scan LinkedList list with iterator itr1
  - For each itr1 position, with element o1, initialize a ListIterator (itr2) at that position.
  - Scan rest of list with itr2, removing elements that equal o1.
- The exception is ConcurrentModificationException, when we go to step itr1 to a new position for the outer loop.
- That itr2 mod was communicated to the list object but its details are kept in the itr2 object, not available to the itr1 object...
Two Iterators: ConcurrentModificationException!

List<Order> list = new LinkedList<Order>(); // or ArrayList
// add some elements to the list: 100, 200, 100, 400
Iterator<Order> outer = list.iterator();
System.out.println("about to do next() in outer loop... list is "+list);
Order o1 = new Order();
System.out.println("outer loop of iterator working on "+o1);
ListIterator<Order> inner = list.listIterator(position + 1);
while (inner.hasNext()) {
    Order o2 = inner.next();
System.out.println("inner loop of iterator working on "+o2);
    if (o1.equals(o2)) {
        System.out.println("removing o2 ="+o2);
        list.remove(); // this works OK
    }
System.out.println("Now advance outer loop of iterator... ");
}
if (inner.hasNext())
    inner.next(); // this throws!!

Remove Duplicates With Two Iterators: Notes

- Note that for this example, the Order object needs an equals method.
- Otherwise, the code would not find duplicates at all.
- We can just test the order id between two Orders.
- We know the list objects are Orders because this is a List<Order>, so have int getId() in the Order API.
- See the code on p. 103 for an example of equals.
- When you code equals, you should also code hashCode.
- Use @Override to make compiler check method really overrides

Order Object for Example, equals by p. 103

```java
public class Order {
    private int id; // unique identifier, basis of equality
    public Order(int id) { this.id = id; }
    public String toString() { return "id: "+id; }
}
```

Safety Removing Duplicates

- Drop the outer iterator, just get() the value of the list element
  into a variable.
- Run the inner loop with an ListIterator over the rest of the list, removing all elements that are equal to that value.
- What to do with huge lists, when using get and/or remove in inner loop means O(n^2) or worse?

Notes on Removing Duplicates From a List

Output:
about to do next() in outer loop... list is [100, 200, 100, 400]
inner loop o2 =100
removing o2 =100
inner loop o2 =400
Now try to advance outer loop of iterator...
Exception in thread "main" java.util.ConcurrentModificationException
at java.base/java.util.LinkedList$ListIterator.checkForComodification(LinkedList.java:97)
at java.base/java.util.LinkedList$ListIterator.next(LinkedList.java:992) at ListRemove.main(ListRemove.java:34)

Safely Removing Duplicates

- Drop the outer iterator, just get the value of the element into a variable.
- Run an internal loop with an iterator, removing all elements that are equal to that value.
- What to do with huge lists, when using get and/or remove in inner loop means O(n^2) or worse?
- Abandon lists!
- You can use HashSet h = new HashSet(); // Set means no dupes, O(n) (good trick for quick deduplication)
- Then put result back in a list, also O(n).
- Another way: toArray, then sort, then pick off unique values, O(n log n)
It's a little like driving a shift car vs. automatic or self
to
get
delete
Queue
a
pop,
an
ADTs
Bags, Queues, and Stacks are covered in S&W
Bags are similar, but do not keep order, but do
But the APIs (listed on page 121) are as expected
In general, we like to use the simplest data structure that
Advanced
peek),
We have seen that a List is pretty complicated, with two
Advanced
JDK
There they are implemented directly in Java, not
other.
Advanced
dequeue
ArrayList
Why use a Queue instead of the List inside it?
often is working
"wide"
Collection
ArrayDeque.
Because it's easier to reason about code that's working with
Advanced
enqueue
is
a Queue
specialized
list
specialized
lists: they keep elements in a kind of order, and tolerate duplicates.
• Bags are similar, but do not keep order, but do
tolerate duplicates
• Bags, Queues, and Stacks are covered in S&W Section 1.3, hopefully covered in cs210
• There they are implemented directly in Java, not using the JDK.
• But the APIs (listed on page 121) are as expected
for classic Queues, Stacks, and Bags.

Specialized Lists

• Queues and Stacks are specialized lists: they keep elements in a kind of order, and tolerate duplicates.
• Bags are similar, but do not keep order, but do tolerate duplicates
• Bags, Queues, and Stacks are covered in S&W Section 1.3, hopefully covered in cs210
• There they are implemented directly in Java, not using the JDK.
• But the APIs (listed on page 121) are as expected
for classic Queues, Stacks, and Bags.

Stacks and Queues: the pure textbook ADTs

A Stack is a specialized List where we insert (push), retrieve
(top or peak), and delete (pop) elements at one end.

A Queue is a specialized list where we insert at one end (enqueue),
retrieve and delete (dequeue) at the other.

Queue Support in the JDK

• There is a Queue<E> interface in the JDK
• The JDK docs list various implementing concrete classes
  including ArrayDeque
• Queue<E> is an ISA Collection<E>, so this is a "wide"
  interface
• In fact, a Queue ISA Deque ISA Collection
• So a Queue is implemented easily with an ArrayDeque.
• We can create a Queue class and implement it with a
  LinkedList or ArrayList to stick to its textbook model.

Linked List/Array List: jack of all trades data structure

• We can see that a LinkedList or ArrayList often is working
  inside another object, providing the needed apparatus
• Why use a Queue instead of the List inside it?
• Because it’s easier to reason about code that’s working with
  a queue than with a List
• We have seen that a List is pretty complicated, with two-way
  iterators and different ways to remove.
• It’s a little like driving a shift car vs. automatic or self-driving
• In general, we like to use the simplest data structure that
  has the capabilities we need.
• Note homework 2 problem on using a LinkedList to
  implement Bag.
Using LinkedList/ArrayList...

- Can we implement a Set<E> with LinkedList<E>?
  - Yes, but it won’t perform like a HashSet or a TreeSet, because of the work of avoiding duplicates, and the cost of traversing the list to determine contains().

- Can we implement a Map<X,Y> with a LinkedList<E>?
  - Yes, use E to hold a Map.Entry<X,Y>. A get will involve a scan of the list, very inefficient.

- So in practice we use LinkedList and ArrayList to implement ’list-like’ data structures, where keeping elements in a certain order can be important, and allowing duplicates.

- FYI: The Lisp language uses lists as the primary data structure.

Priority Queue: another specialized list, sort of

- Often data are processed in a specific order, the priority
  - Examples
    - In OS process scheduling, a process comes with a priority, and high priority processes are executed earlier than low priority ones
    - In network routing, high priority traffic (VoIP, IPTV) are delivered before other traffic

- We need to dynamically maintain the pending jobs
  - New jobs keep coming in
    - When a server becomes available, the highest-priority job is removed from PQ and serviced
  - We’ll defer studying Priority Queues until later in the term.