List

...

NOTE

We use List as illustrative example of collections

Study the others from textbook!

What is a List?

- A List is an ordered Collection (sometimes called a sequence)
- Lists may contain duplicate elements
- Features all operations inherited from Collection…

…then some more…

Collection Operations

They all work just as expected w/ a few remarks;

remove(…)

- Always removes the first occurrence

add(…) & addAll(…)

- Always append at the end of the List

```
// appends list2 at end of list1
list1.addAll(list2);
```

what about a non-destructive way?

```
// returns a new list
List<Type> list3 = new
            ArrayList<Type>(list1);
list3.addAll(list2);
```

Remember the conversion constructors?

… Additional features

Positional access

- manipulates elements based on their index; e.g. get, set, add, addAll & remove

Search

- searches for specified object in list and returns its index; e.g. indexOf, lastIndexOf

Iteration

- extends Iterator semantics to leverage list's sequential nature; i.e. listIterator

Range-view

- The sublist method performs arbitrary range operations on the list

Positional Access & Search Operations

get(…), add(…)

- Just work as expected

set(…), remove(…)

- return the old value that is being overwritten or removed

indexOf(…), lastIndexOf(…)

- return the first or last index of the specified element in the list

Positional Access & Search Operations

addAll(…)

- inserts all the elements of specified Collection starting at specified position
- elements are inserted in the order they are returned by the iterator of the Collection specified as parameter
- This is the positional access analog of Collection's addAll operation

List Interface Implementations

Java offers 2 general-purpose List implementations;

ArrayList

- which is usually the better-performing implementation

LinkedList

- which offers better performance under certain circumstances

…which ones?

When to use each implementation

Operation	LinkedList	ArrayList
get(int index)	O(n/4) average	**O(1)**
add(E element)	O(1)	O(1) but O(n) worst case if array needs to be resized
add(int index, E element)	**O(n/4) average** (start @ end or front, whichever is closer) **but O(1) when index==0**	O(n/2) average (need shifting)
remove(int index)	O(n/4) average	O(n/2) average
Iterator.remove()	**O(1)**	O(n/2) average
ListIterator.add(E element)	**O(1)**	O(n/2) average
NOTES	O(n/4) is average O(1) best case (e.g. index = 0) O(n/2) worst case (middle of list)	O(n/2) is average O(1) best case (end of list) O(n) worst case (start of list)

stack overflow http://stackoverflow.com/questions/322715/when-to-use-linkedlist-over-arraylist

Let's take a quick example…

all permutations occur with equal likelihood, assuming an unbiased source of randomness

```
// swap two indexed values in a List
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}

// use this to shuffle any List
public static void shuffle(List<?> list, Random rnd) {
    for (int i = list.size(); i > 1; i--)
        swap(list, i - 1, rnd.nextInt(i));
}

// Java Collections Framework shuffling is efficient & fair
```

requiring exactly list.size()-1 swaps

Let's use this shuffler

```
// Display the words passed at the CLI in random order

public class Shuffle {
      public static void main(String[] args) {
            List<String> list = new ArrayList<String>();
            for (String a : args){
                  list.add(a);
            }
            Collections.shuffle(list, new Random());
            System.out.println(list);
      }
}
```

… now we refactor ☺

```
public class Shuffle {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        for (String a : args)
            list.add(a);
        Collections.shuffle(list, new Random());
        System.out.println(list);
    }
}
```

```
public class Shuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```

- Not resizable
- No add / remove

Iterating on Lists w/ Iterators – 2 ways

Iterator

- This iterator object returns the elements of the list in proper sequence

ListIterator

- Richer iterator which allows you to…
- …traverse list in either direction
- …Modifies list during iteration
- …Obtains current position of the iterator

Features of each Iterator

hasNext(…), next(…), remove(…)

- Inherited by ListIterator from Iterator
- do exactly the same thing in both interfaces

hasPrevious(…), previous(…)

- exact analogues of hasNext(…) and next(…)
- hasPrevious(…) → refers to element before the (implicit) cursor
- hasNext(…) → refers to the element after the cursor
- Previous(…) → moves the cursor backward
- next(…) → moves it forward

Example – Iterating backward

```
for (     ListIterator<Type> it = list.listIterator(list.size());
          it.hasPrevious();
     ) {

     Type t = it.previous();

     ...
}
```

Two forms of listIterator			
#1	w/o arguments	returns ListIterator positioned at	beginning of the list
#2	w/ int argument	returns ListIterator positioned at	specified index

Example – Replacing w/ Iterator

```
public static <E> void replace(List<E> list, E val, E newVal) {

      for (ListIterator<E> it = list.listIterator();
            it.hasNext();
      ){
            if (val==null   ? it.next()==null
                            : val.equals(it.next())){
                  it.set(newVal);
            }
      }
}
```

Null Pointer Exception
if we did not handle
differently val being null

Example – Replacing 1 by many w/ Iterator

```
public static <E>
void replace(List<E> list, E val, List<? extends E> newVals){

     for (ListIterator<E> it = list.listIterator();
          it.hasNext(); ){
          if (val==null  ? it.next()==null
                         : val.equals(it.next())){
               it.remove();
               for (E e : newVals) it.add(e);
          }
     }
}
```