CSE 373

Java Collection Framework, Part 2: Priority Queue, Map

slides created by Marty Stepp
http://www.cs.washington.edu/373/

Priority queue ADT

- **priority queue**: a collection of ordered elements that provides fast access to the minimum (or maximum) element
  - usually implemented using a tree structure called a *heap*

- priority queue operations:
  - `add` adds in order; O(log *N*) worst
  - `peek` returns **minimum** value; O(1) always
  - `remove` removes/returns **minimum** value; O(log *N*) worst
  - `isEmpty`, `clear`, `size`, `iterator` O(1) always

Java's `PriorityQueue` class

```
public class PriorityQueue<E> implements Queue<E>
```

Method/Constructor	Description	Runtime
`PriorityQueue<E>()`	constructs new empty queue	O(1)
`add(E value)`	adds value in sorted order	O(log *N*)
`clear()`	removes all elements	O(1)
`iterator()`	returns iterator over elements	O(1)
`peek()`	returns minimum element	O(1)
`remove()`	removes/returns min element	O(log *N*)

```
Queue<String> pq = new PriorityQueue<String>();
pq.add("Stuart");
pq.add("Marty");
...
```

Priority queue ordering

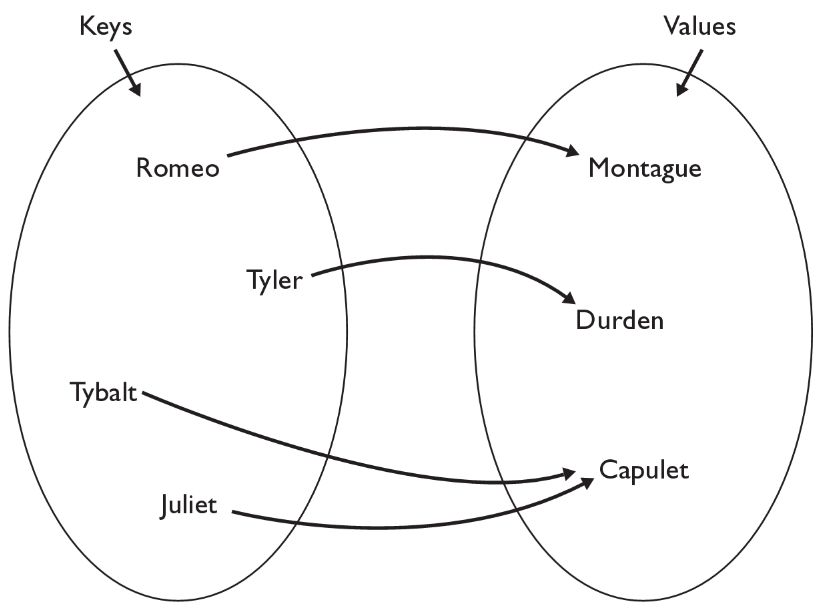
- For a priority queue to work, elements must have an ordering
  - in Java, this means implementing the `Comparable` interface
    - many existing types (Integer, String, etc.) already implement this
    - if you store objects of your own types in a PQ, you must implement it
  - `TreeSet` and `TreeMap` also require `Comparable` types

```
public class Foo implements Comparable<Foo> {
    …
    public int compareTo(Foo other) {
        // Return > 0 if this object is > other
        // Return < 0 if this object is < other
        // Return   0 if this object == other
    }
}
```

The Map ADT

- **map**: Holds a set of unique *keys* and a collection of *values*, where each key is associated with one value.
  - a.k.a. "dictionary", "associative array", "hash"

- basic map operations:
  - **put**(*key*, *value*): Adds a mapping from a key to a value.
  - **get**(*key*): Retrieves the value mapped to the key.
  - **remove**(*key*): Removes the given key and its mapped value.



`myMap.get("Juliet")` returns `"Capulet"`

Map concepts

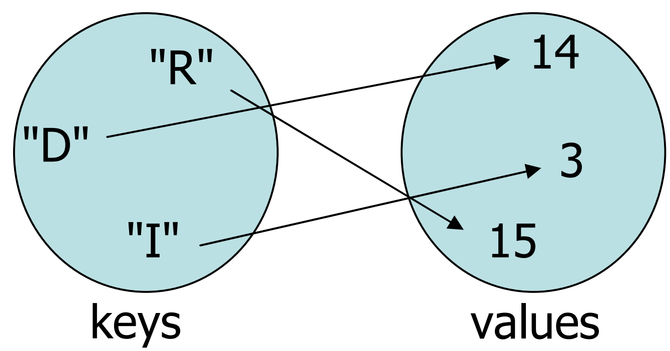
- a map can be thought of as generalization of a tallying array
  - the "index" (key) doesn't have to be an `int`
  - count digits: `22092310907`

index	0	1	2	3	4	5	6	7	8	9
value	3	1	3	0	0	0	0	1	0	2

```
// (R)epublican, (D)emocrat, (I)ndependent
```

  - count votes: `"RDDDDRRRRRDDDDDDRDRRIRDRRIRDRRID"`

key	"R"	"D"	"I"
value	15	14	3



`Map` implementation

- in Java, maps are represented by `Map` interface in `java.util`

- `Map` is implemented by the `HashMap` and `TreeMap` classes
  - `HashMap`: implemented using an array called a "hash table"; extremely fast: O(1) ; keys are stored in unpredictable order
  - `TreeMap`: implemented as a linked "binary tree" structure; very fast: O(log N) ; keys are stored in sorted order

  - A map requires 2 type parameters: one for keys, one for values.

```
// maps from String keys to Integer values
Map<String, Integer> votes = new HashMap<String, Integer>();
```

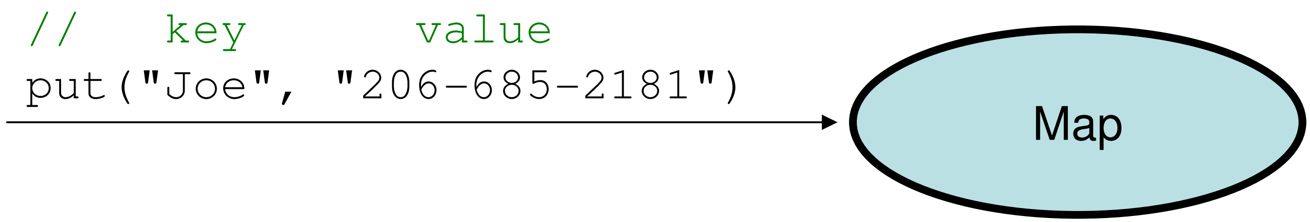
Map methods

`put(`**key**`, `**value**`)`	adds a mapping from the given key to the given value; if the key already exists, replaces its value with the given one
`get(`**key**`)`	returns the value mapped to the given key (`null` if not found)
`containsKey(`**key**`)`	returns `true` if the map contains a mapping for the given key
`remove(`**key**`)`	removes any existing mapping for the given key
`clear()`	removes all key/value pairs from the map
`size()`	returns the number of key/value pairs in the map
`isEmpty()`	returns `true` if the map's size is 0
`toString()`	returns a string such as `"{a=90, d=60, c=70}"`

`keySet()`	returns a set of all keys in the map
`values()`	returns a collection of all values in the map
`putAll(`**map**`)`	adds all key/value pairs from the given map to this map
`equals(`**map**`)`	returns `true` if given map has the same mappings as this one

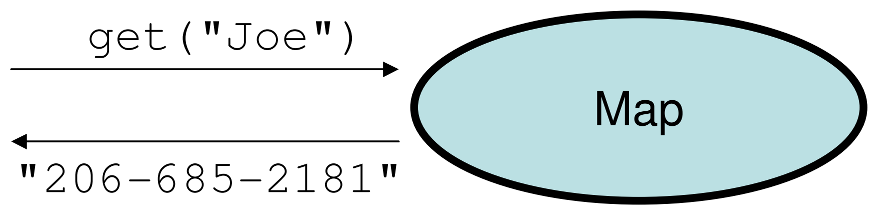
Using maps

- A map allows you to get from one half of a pair to the other.
  - Remembers one piece of information about every index (key).



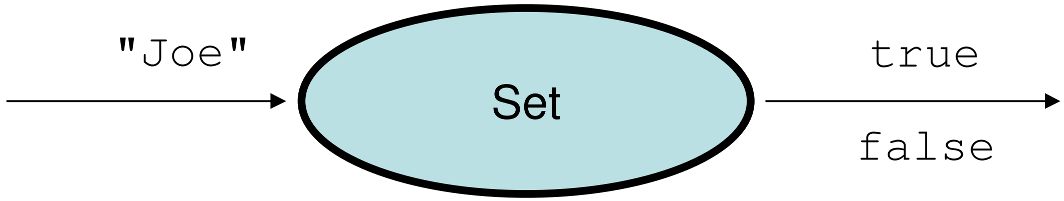
  - Later, we can supply only the key and get back the related value:

    Allows us to ask: *What is Joe's phone number?*

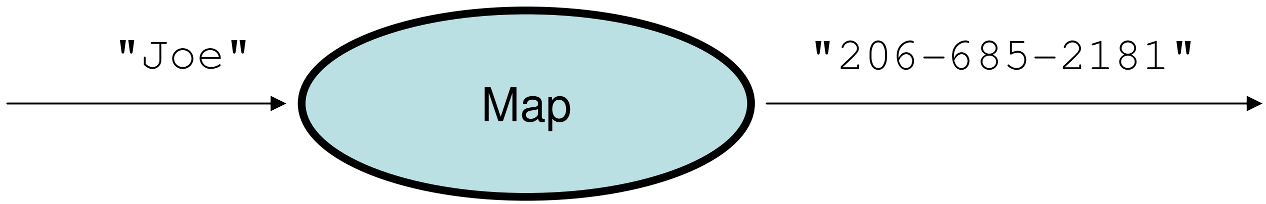


Maps vs. sets

- A set is like a map from elements to `boolean` values.
  - *Set: Is Joe found in the set? (true/false)*



  - *Map: What is Joe's phone number?*



keySet and values

- `keySet` method returns `Set` of all keys in map
  - can loop over the keys in a foreach loop
  - can get each key's associated value by calling `get` on the map

```
Map<String, Integer> ages = new TreeMap<String, Integer>();
ages.put("Joe", 57);
ages.put("Geneva", 2);  // ages.keySet() returns Set<String>
ages.put("Vicki", 19);
for (String name : ages.keySet()) {           // Geneva -> 2
    int age = ages.get(name);                  // Joe -> 57
    System.out.println(name + " -> " + age); // Vicki -> 19
}
```

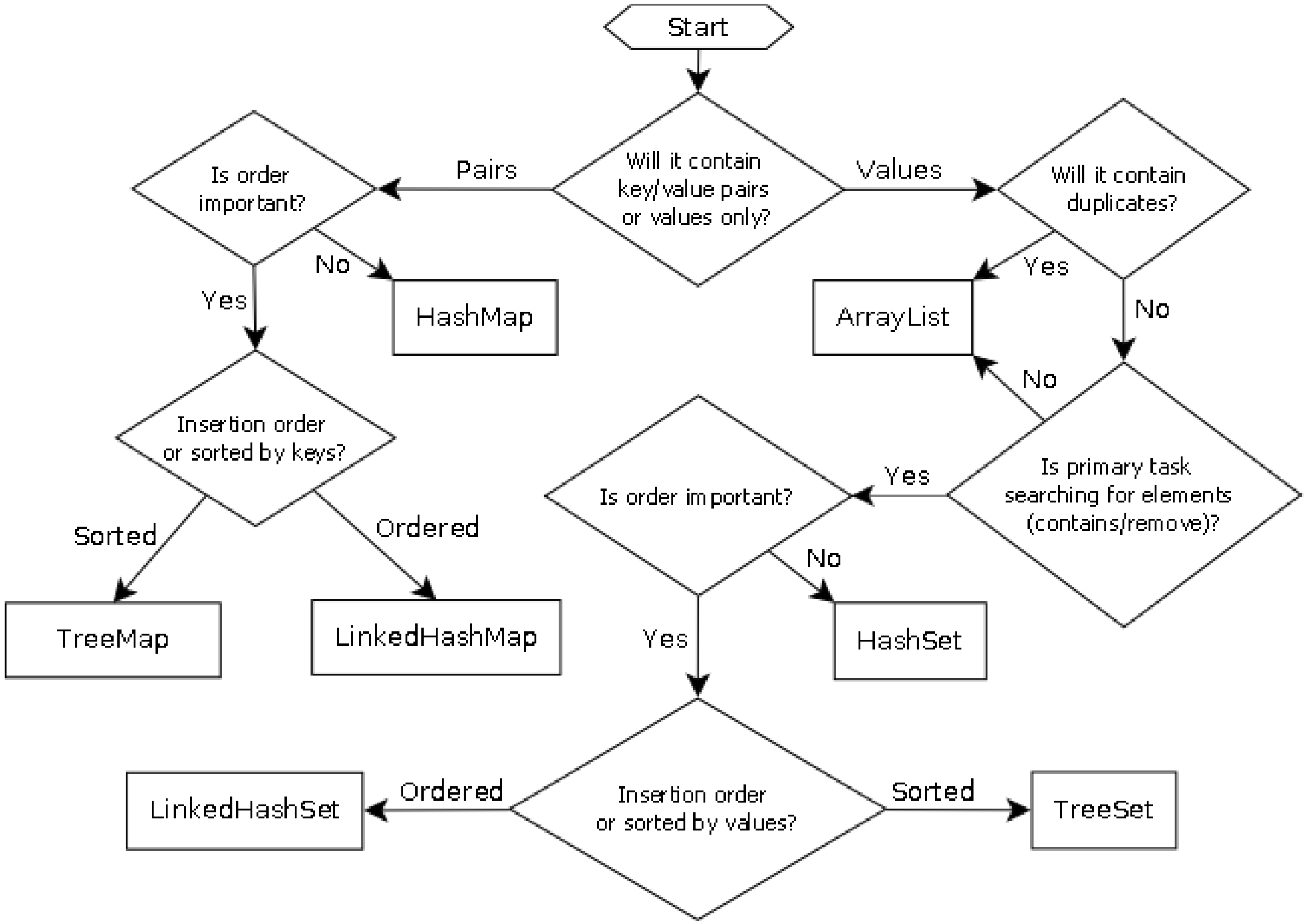
- `values` method returns `Collection` of all values in map
  - `ages.values()` above returns `[2, 57, 19]`
  - can loop over the values with a for-each loop
  - no easy way to get from a value back to its associated key(s)

Collections summary

collection	ordering	benefits	weaknesses
array	by index	fast; simple	little functionality; cannot resize
`ArrayList`	by insertion, by index	random access; fast to modify at end	slow to modify in middle/front
`LinkedList`	by insertion, by index	fast to modify at both ends	poor random access
`TreeSet`	sorted order	sorted; O(log N)	must be comparable
`HashSet`	unpredictable	very fast; O(1)	unordered
`LinkedHashSet`	order of insertion	very fast; O(1)	uses extra memory
`TreeMap`	sorted order	sorted; O(log N)	must be comparable
`HashMap`	unpredictable	very fast; O(1)	unordered
`LinkedHashMap`	order of insertion	very fast; O(1)	uses extra memory
`PriorityQueue`	natural/comparable	fast ordered access	must be comparable

- It is important to be able to choose a collection properly based on the capabilities needed and constraints of the problem to solve.

Choosing a collection



- see also: http://initbinder.com/bunker/wp-content/uploads/2011/03/collections.png

Compound collections

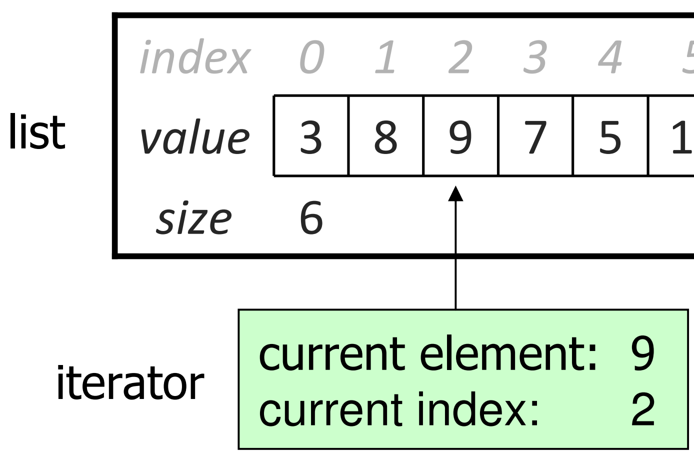
- You will often find that you want a collection of collections:
  - a list of lists; a map of strings to lists; a queue of sets; ...

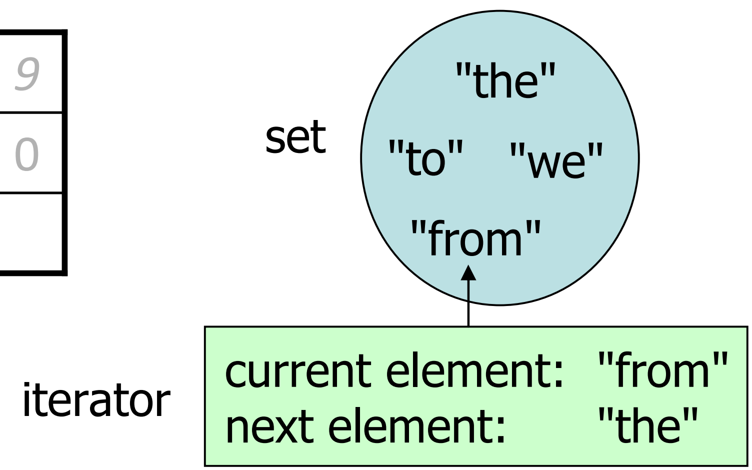
- *Example:* how would you store people's friends?
  - i.e., I need to quickly look up the names of all of Jimmy's buddies, or test whether a given person is a friend of Jimmy's or not.

```
// don't forget to initialize each Set of friends
Map<String, Set<String>> pals =
        new HashMap<String, Set<String>>();
pals.put("Jimmy", new HashSet<String>());
pals.get("Jimmy").add("Bill");
pals.get("Jimmy").add("Katherine");
pals.get("Jimmy").add("Stuart");
```

Iterators (11.1)

- **iterator**: An object that allows a client to traverse the elements of any collection.
  - Remembers a position, and lets you:
    - get the element at that position
    - advance to the next position
    - remove the element at that position





`Iterator` methods

`hasNext()`	returns `true` if there are more elements to examine
`next()`	returns the next element from the collection (throws a `NoSuchElementException` if there are none left to examine)
`remove()`	removes the last value returned by `next()` (throws an `IllegalStateException` if you haven't called `next()` yet)

- `Iterator` interface in `java.util`
  - every collection has an `iterator()` method that returns an iterator over its elements

```
Set<String> set = new HashSet<String>();
...
Iterator<String> itr = set.iterator();
...
```

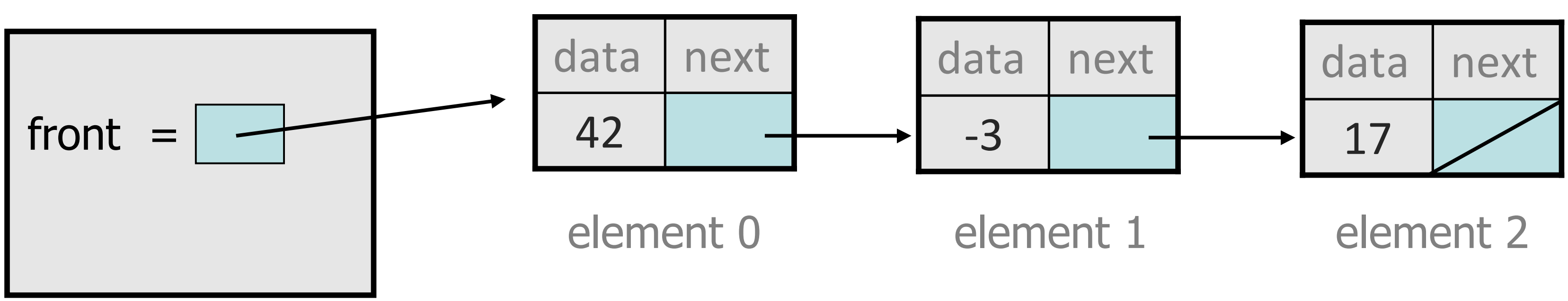
Iterator example

```
Set<Integer> scores = new TreeSet<Integer>();
scores.add(94);
scores.add(38);   // Jenny
scores.add(87);
scores.add(43);   // Marty
scores.add(72);
...

Iterator<Integer> itr = scores.iterator();
while (itr.hasNext()) {
    int score = itr.next();

    System.out.println("The score is " + score);

    // eliminate any failing grades
    if (score < 60) {
        itr.remove();
    }
}
System.out.println(scores);  // [72, 87, 94]
```

A surprising example

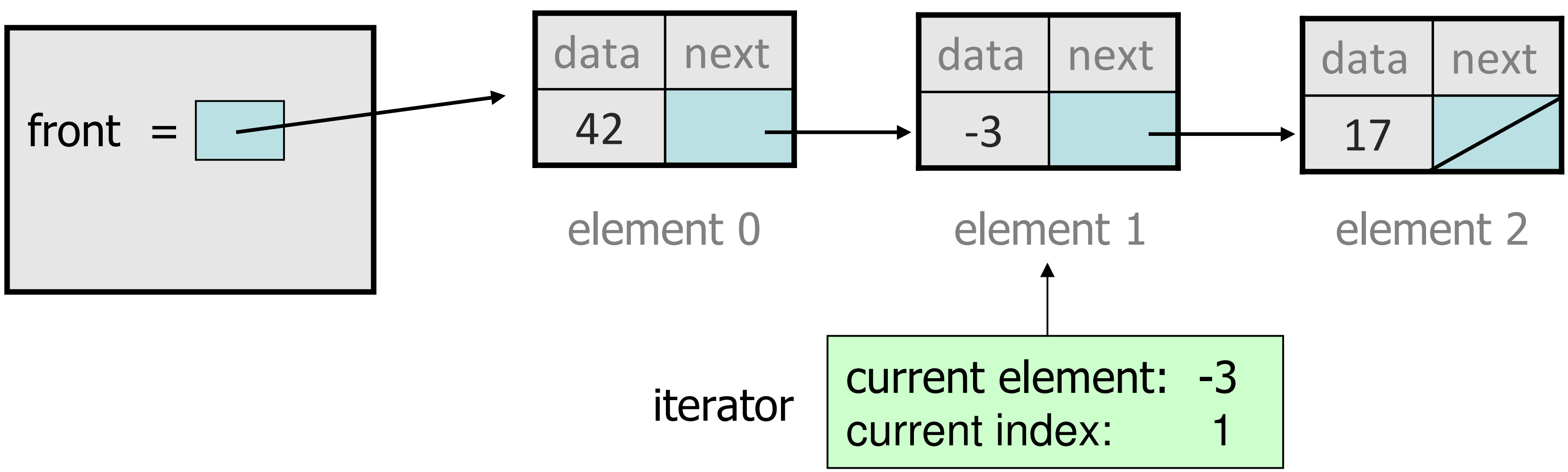
- What's bad about this code?

```
List<Integer> list = new LinkedList<Integer>();
... (add lots of elements) ...
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```



Iterators and linked lists

- Iterators are particularly useful with linked lists.
  - The previous code is $O(N^2)$ because each call on `get` must start from the beginning of the list and walk to index `i`.
  - Using an iterator, the same code is $O(N)$. The iterator remembers its position and doesn't start over each time.



ListIterator

`add(`**value**`)`	inserts an element just after the iterator's position
`hasPrevious()`	`true` if there are more elements *before* the iterator
`nextIndex()`	the index of the element that would be returned the next time `next` is called on the iterator
`previousIndex()`	the index of the element that would be returned the next time `previous` is called on the iterator
`previous()`	returns the element before the iterator (throws a `NoSuchElementException` if there are none)
`set(`**value**`)`	replaces the element last returned by `next` or `previous` with the given value

```
ListIterator<String> li = myList.listIterator();
```

- lists have a more powerful `ListIterator` with more methods
  - can iterate forwards or backwards
  - can add/set element values (efficient for linked lists)