CSE 143
Lecture 8

Iterators; Comparable

reading: 11.2; 10.2

slides created by Marty Stepp
http://www.cs.washington.edu/143/

Examining sets and maps

- elements of Java `Set`s and `Map`s can't be accessed by index
  - must use a "foreach" loop:

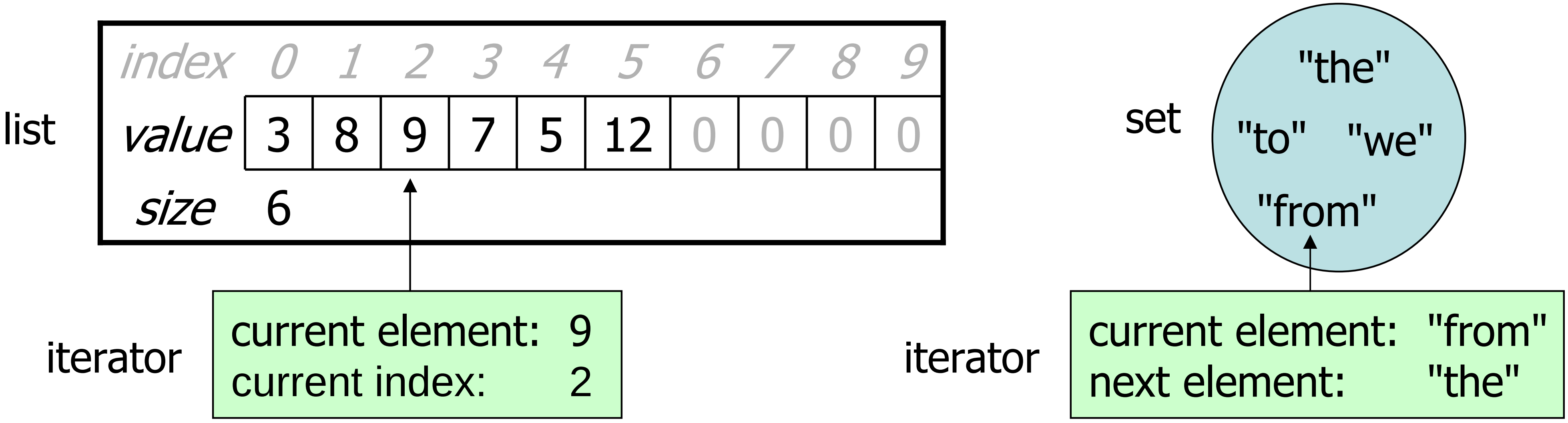
```
Set<Integer> scores = new HashSet<Integer>();
for (int score : scores) {
    System.out.println("The score is " + score);
}
```

  - Problem: foreach is read-only; cannot modify set while looping

```
for (int score : scores) {
    if (score < 60) {
    // throws a ConcurrentModificationException
        scores.remove(score);
    }
}
```

Iterators (11.1)

- **iterator**: An object that allows a client to traverse the elements of any collection, regardless of its implementation.
  - Remembers a position within a collection, and allows you to:
    - get the element at that position
    - advance to the next position
    - (possibly) remove or change the element at that position
  - Benefit: A common way to examine *any* collection's elements.



list

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	3	8	9	7	5	12	0	0	0	0
size	6									

iterator

current element: 9
current index: 2

set

"the"
"to" "we"
"from"

iterator

current element: "from"
next element: "the"

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 from the collection the last value returned by `next()` (throws `IllegalStateException` if you have not 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 HashSet<Integer>();
scores.add(38);
scores.add(94);
scores.add(87);
scores.add(43);
scores.add(62);
…

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);  // [62, 94, 87]
```

Iterator example 2

```
Map<String, Integer> scores = new HashMap<String, Integer>();
scores.put("Kim", 38);
scores.put("Lisa", 94);
scores.put("Ryan", 87);
scores.put("Morgan", 43);
scores.put("Marisa", 62);
…

Iterator<String> itr = scores.keySet().iterator();
while (itr.hasNext()) {
    String name = itr.next();
    int score = scores.get(name);
    System.out.println(name + " got " + score);

    // eliminate any failing students
    if (score < 60) {
        itr.remove();     // removes name and score
    }
}
System.out.println(scores);  // {Marisa=62, Lisa=94, Ryan=87}
```

Exercise

- Modify the Book Search program from last lecture to eliminate any words that are plural or all-uppercase from the collection.

Set/Map and ordering

- Some types have a notion of a *natural ordering* .
  - TreeSet/Map store values sorted by their natural ordering.

```
Set<Integer> scores = new HashSet<Integer>();
scores.add(38);
scores.add(94);
scores.add(87);
scores.add(43);              // unpredictable order
scores.add(62);
System.out.println(scores);  // [62, 94, 43, 87, 38]
```

```
Set<Integer> scores = new TreeSet<Integer>();
scores.add(38);
scores.add(94);
scores.add(87);
scores.add(43);              // sorted natural order
scores.add(62);
System.out.println(scores);  // [38, 43, 62, 87, 94]
```

Ordering our own types

- We cannot make a `TreeSet` or `TreeMap` of any arbitrary type, because Java doesn't know how to order the elements.
  - The program compiles but crashes when we run it.

```
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));
tags.add(new HtmlTag("b", false));
...

Exception in thread "main" java.lang.ClassCastException
        at java.util.TreeMap.put(TreeMap.java:542)
        at java.util.TreeSet.add(TreeSet.java:238)
        at MyProgram.main(MyProgram.java:24)
```

Comparable (10.2)

```
public interface Comparable<E> {
    public int compareTo(E other);
}
```

- A class can implement the `Comparable` interface to define a natural ordering function for its objects.

- A call of `a.compareTo(b)` should return:

  a value < 0 if a comes "before" b in the ordering,
  a value > 0 if a comes "after" b in the ordering,
  or 0 if a and b are considered "equal" in the ordering.

Comparable example

```
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    …

    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;   // same x, smaller y
        } else if (y > other.y) {
            return 1;    // same x, larger y
        } else {
            return 0;    // same x and same y
        }
    }
}
```

compareTo tricks

- subtraction trick - Subtracting related numeric values produces the right result for what you want `compareTo` to return:

```
// sort by x and break ties by y
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;   // different x
    } else {
        return y - other.y;   // same x; compare y
    }
}
```

  - The idea:
    - if `x > other.x`, then `x - other.x > 0`
    - if `x < other.x`, then `x - other.x < 0`
    - if `x == other.x`, then `x - other.x == 0`

`compareTo` tricks 2

- delegation trick - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

- `toString` trick - If your object's `toString` representation is related to the ordering, use that to help you:

```
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```

Comparable and sorting

- The `Arrays` and `Collections` classes in `java.util` have a static method `sort` that sorts the elements of an array/list

```
Point[] points = new Point[3];
points[0] = new Point(7, 6);
points[1] = new Point(10, 2)
points[2] = new Point(7, -1);
points[3] = new Point(3, 11);
Arrays.sort(points);
System.out.println(Arrays.toString(points));
// (3, 11), (7, -1), (7, 6), (10, 2)


List<Point> points = new ArrayList<Point>();
points.add(new Point(7, 6));
…
Collections.sort(points);
System.out.println(points);
// (3, 11), (7, -1), (7, 6), (10, 2)
```

Arrays class

Method name	Description
`asList(`**value1**`, `**...**`, `**valueN**`)`	returns a `List` containing the given values as its elements
`binarySearch(`**array**`, `**value**`)`	returns the index of the given value in a sorted array (< 0 if not found)
`copyOf(`**array**`)`	returns a new array with same elements
`equals(`**array1**`, `**array2**`)`	returns `true` if the two arrays contain the same elements in the same order
`fill(`**array**`, `**value**`)`	sets every element to have given value
`sort(`**array**`)`	arranges elements into ascending order
`toString(`**array**`)`	returns a string representing the array, such as `"[10, 30, 17]"`

Collections class

Method name	**Description**
binarySearch(**list**, **value**)	returns the index of the given value in a sorted list (< 0 if not found)
copy(**listTo**, **listFrom**)	copies **listFrom**'s elements to **listTo**
emptyList(), emptyMap(), emptySet()	returns a read-only collection of the given type that has no elements
fill(**list**, **value**)	sets every element in the list to have the given value
max(**collection**), min(**collection**)	returns largest/smallest element
replaceAll(**list**, **old**, **new**)	replaces an element value with another
reverse(**list**)	reverses the order of a list's elements
shuffle(**list**)	arranges elements into a random order
sort(**list**)	arranges elements into ascending order