

Building Java Programs

Chapter 10

ArrayList

Copyright (c) Pearson 2013.
All rights reserved.

Java Collections

Reading

- Building Java Programs, Ch. 10

Learning Outcomes

- ArrayList operations
- Wrapper classes
- Generics
- Ordering & comparison
- LinkedList objects
- Iterators
- Set objects
- Map objects

Summary

- Java's **ArrayList** supports a flexible sized array of objects
- ArrayList is a **generic** class. You must specify the data type when creating an ArrayList instance
- ArrayList supports various operations to retrieve / modify / search for elements
- Collections support **iteration** via **for-each** loop
- Collections storing primitive values (e.g. int or double) must be created with a **wrapper** type (e.g. Integer, Double)
- Objects stored in an ArrayList must implement the **comparable** interface to allow comparison and sorting

Exercise

- Write a program that reads a file and displays the words of that file as a list.
 - First display all words.
 - Then display them with all plurals (ending in "s") capitalized.
 - Then display them in reverse order.
 - Then display them with all plural words removed.
- Should we solve this problem using an array?
 - Why or why not?

Naive solution

```
String[] allWords = new String[1000];  
int wordCount = 0;
```

```
Scanner input = new Scanner(new File("data.txt"));  
while (input.hasNext()) {  
    String word = input.next();  
    allWords[wordCount] = word;  
    wordCount++;  
}
```

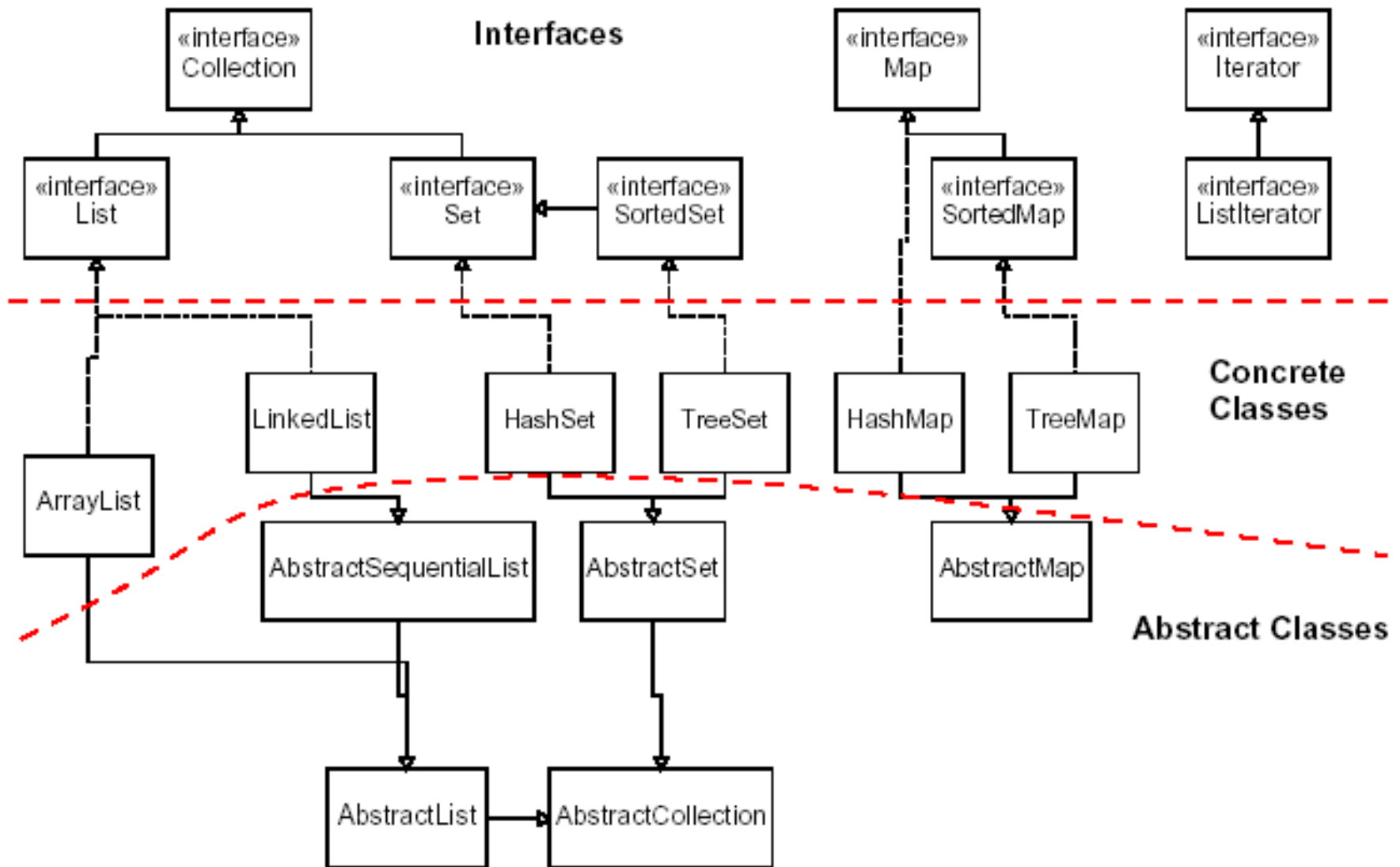
- Problem: You don't know how many words the file will have.
 - Hard to create an array of the appropriate size.
 - Later parts of the problem are more difficult to solve.
- Luckily, there are other ways to store data besides in an array.

Collections

- **collection**: an object that stores data; a.k.a. "data structure"
 - the objects stored are called **elements**
 - some collections maintain an ordering; some allow duplicates
 - typical operations: add, remove, clear, contains (search), size
 - examples found in the Java class libraries:
 - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`
 - all collections are in the `java.util` package

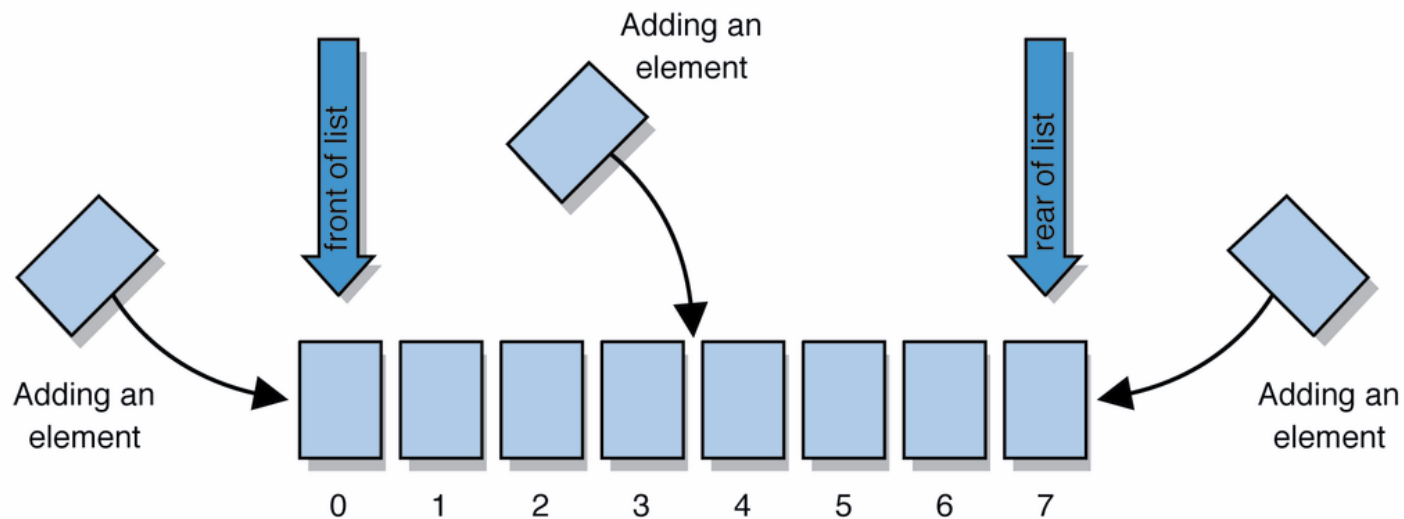
```
import java.util.*;
```

Java collections framework



Lists

- **list**: a collection storing an ordered sequence of elements
 - each element is accessible by a 0-based **index**
 - a list has a **size** (number of elements that have been added)
 - elements can be added to the front, back, or elsewhere
 - in Java, a list can be represented as an **ArrayList** object



Idea of a list

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

`[]`

- You can add items to the list.
 - The default behavior is to add to the end of the list.

`[hello, ABC, goodbye, okay]`

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
 - Think of an "array list" as an automatically resizing array object.
 - Internally, the list is implemented using an array and a size field.

ArrayList methods (10.1)

<code>add(value)</code>	appends value at end of list
<code>add(index, value)</code>	inserts given value just before the given index, shifting subsequent values to the right
<code>clear()</code>	removes all elements of the list
<code>indexOf(value)</code>	returns first index where given value is found in list (-1 if not found)
<code>get(index)</code>	returns the value at given index
<code>remove(index)</code>	removes/returns value at given index, shifting subsequent values to the left
<code>set(index, value)</code>	replaces value at given index with given value
<code>size()</code>	returns the number of elements in list
<code>toString()</code>	returns a string representation of the list such as "[3, 42, -7, 15]"

ArrayList methods 2

<code>addAll(list)</code> <code>addAll(index, list)</code>	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
<code>contains(value)</code>	returns true if given value is found somewhere in this list
<code>containsAll(list)</code>	returns true if this list contains every element from given list
<code>equals(list)</code>	returns true if given other list contains the same elements
<code>iterator()</code> <code>listIterator()</code>	returns an object used to examine the contents of the list (seen later)
<code>lastIndexOf(value)</code>	returns last index value is found in list (-1 if not found)
<code>remove(value)</code>	finds and removes the given value from this list
<code>removeAll(list)</code>	removes any elements found in the given list from this list
<code>retainAll(list)</code>	removes any elements not found in given list from this list
<code>subList(from, to)</code>	returns the sub-portion of the list between indexes from (inclusive) and to (exclusive)
<code>toArray()</code>	returns the elements in this list as an array

Type Parameters (Generics)

```
ArrayList<Type> name = new ArrayList<Type>();
```

- When constructing an `ArrayList`, you must specify the type of elements it will contain between `<` and `>`.
 - This is called a type parameter or a generic class.
 - Allows the same `ArrayList` class to store lists of different types.

```
ArrayList<String> names = new ArrayList<String>();  
names.add("Marty Stepp");  
names.add("Stuart Reges");
```

Learning about classes

- The [Java API Specification](http://java.sun.com/javase/6/docs/api/) is a huge web page containing documentation about every Java class and its methods.
 - The link to the API Specs is on the course web site.



ArrayList vs. array

- construction

```
String[] names = new String[5];  
ArrayList<String> list = new ArrayList<String>();
```

- storing a value

```
names[0] = "Jessica";  
list.add("Jessica");
```

- retrieving a value

```
String s = names[0];  
String s = list.get(0);
```

ArrayList vs. array 2

- doing something to each value that starts with "B"

```
for (int i = 0; i < names.length; i++) {  
    if (names[i].startsWith("B")) { ... }  
}
```

```
for (int i = 0; i < list.size(); i++) {  
    if (list.get(i).startsWith("B")) { ... }  
}
```

- seeing whether the value "Benson" is found

```
for (int i = 0; i < names.length; i++) {  
    if (names[i].equals("Benson")) { ... }  
}
```

```
if (list.contains("Benson")) { ... }
```

Exercise, revisited

- Write a program that reads a file and displays the words of that file as a list.
 - First display all words.
 - Then display them in reverse order.
 - Then display them with all plurals (ending in "s") capitalized.
 - Then display them with all plural words removed.

Exercise solution (partial)

```
ArrayList<String> allWords = new ArrayList<String>();  
Scanner input = new Scanner(new File("words.txt"));  
while (input.hasNext()) {  
    String word = input.next();  
    allWords.add(word);  
}  
System.out.println(allWords);  
  
// remove all plural words  
for (int i = 0; i < allWords.size(); i++) {  
    String word = allWords.get(i);  
    if (word.endsWith("s")) {  
        allWords.remove(i);  
        i--;  
    }  
}  
}
```

ArrayList as parameter

```
public static void name(ArrayList<Type> name) {
```

- Example:

```
// Removes all plural words from the given list.
```

```
public static void removePlural(ArrayList<String> list) {  
    for (int i = 0; i < list.size(); i++) {  
        String str = list.get(i);  
        if (str.endsWith("s")) {  
            list.remove(i);  
            i--;  
        }  
    }  
}
```

- You can also return a list:

```
public static ArrayList<Type> methodName(params)
```

ArrayList of primitives?

- The type you specify when creating an `ArrayList` must be an object type; it cannot be a primitive type.

```
// illegal -- int cannot be a type parameter  
ArrayList<int> list = new ArrayList<int>();
```

- But we can still use `ArrayList` with primitive types by using special classes called wrapper classes in their place.

```
// creates a list of ints  
ArrayList<Integer> list = new ArrayList<Integer>();
```

Wrapper classes



Primitive Type	Wrapper Type
int	Integer
double	Double
char	Character
boolean	Boolean

- A wrapper is an object whose sole purpose is to hold a primitive value.
- Once you construct the list, use it with primitives as normal:

```
ArrayList<Double> grades = new ArrayList<Double>();  
grades.add(3.2);  
grades.add(2.7);  
...  
double myGrade = grades.get(0);
```

Exercise

- Write a program that reads a file full of numbers and displays all the numbers as a list, then:
 - Prints the average of the numbers.
 - Prints the highest and lowest number.
 - Filters out all of the even numbers (ones divisible by 2).

Exercise solution (partial)

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
Scanner input = new Scanner(new File("numbers.txt"));
while (input.hasNextInt()) {
    int n = input.nextInt();
    numbers.add(n);
}
System.out.println(numbers);
filterEvens(numbers);
System.out.println(numbers);
...

// Removes all elements with even values from the given list.
public static void filterEvens(ArrayList<Integer> list) {
    for (int i = list.size() - 1; i >= 0; i--) {
        int n = list.get(i);
        if (n % 2 == 0) {
            list.remove(i);
        }
    }
}
```

Other Exercises

- Write a method `reverse` that reverses the order of the elements in an `ArrayList` of strings.
- Write a method `capitalizePlurals` that accepts an `ArrayList` of strings and replaces every word ending with an "s" with its uppercased version.
- Write a method `removePlurals` that accepts an `ArrayList` of strings and removes every word in the list ending with an "s", case-insensitively.

Out-of-bounds

- Legal indexes are between **0** and the **list's size() - 1**.
 - Reading or writing any index outside this range will cause an `IndexOutOfBoundsException`.

```
ArrayList<String> names = new ArrayList<String>();  
names.add("Marty");    names.add("Kevin");  
names.add("Vicki");    names.add("Larry");  
System.out.println(names.get(0));           // okay  
System.out.println(names.get(3));           // okay  
System.out.println(names.get(-1));         // exception  
names.add(9, "Aimee");                     // exception
```

index	0	1	2	3
value	Marty	Kevin	Vicki	Larry

ArrayList "mystery"

```
ArrayList<Integer> list = new ArrayList<Integer>();  
for (int i = 1; i <= 10; i++) {  
    list.add(10 * i);    // [10, 20, 30, 40, ..., 100]  
}
```

- What is the output of the following code?

```
for (int i = 0; i < list.size(); i++) {  
    list.remove(i);  
}  
System.out.println(list);
```

- Answer:

[20, 40, 60, 80, 100]

ArrayList "mystery" 2

```
ArrayList<Integer> list = new ArrayList<Integer>();  
for (int i = 1; i <= 5; i++) {  
    list.add(2 * i);    // [2, 4, 6, 8, 10]  
}
```

- What is the output of the following code?

```
int size = list.size();  
for (int i = 0; i < size; i++) {  
    list.add(i, 42);    // add 42 at index i  
}  
System.out.println(list);
```

- Answer:

```
[42, 42, 42, 42, 42, 2, 4, 6, 8, 10]
```

ArrayList as parameter

```
public static void name(ArrayList<Type> name) {
```

- Example:

```
// Removes all plural words from the given list.
```

```
public static void removePlural(ArrayList<String> list) {  
    for (int i = 0; i < list.size(); i++) {  
        String str = list.get(i);  
        if (str.endsWith("s")) {  
            list.remove(i);  
            i--;  
        }  
    }  
}
```

- You can also return a list:

```
public static ArrayList<Type> methodName(params)
```

Exercise

- Write a method `addStars` that accepts an array list of strings as a parameter and places a `*` after each element.
 - Example: if an array list named `list` initially stores:
`[the, quick, brown, fox]`
 - Then the call of `addStars(list);` makes it store:
`[the, *, quick, *, brown, *, fox, *]`
- Write a method `removeStars` that accepts an array list of strings, assuming that every other element is a `*`, and removes the stars (undoing what was done by `addStars` above).

Exercise solution

```
public static void addStars(ArrayList<String> list) {  
    for (int i = 0; i < list.size(); i += 2) {  
        list.add(i, "*");  
    }  
}  
  
public static void removeStars(ArrayList<String> list) {  
    for (int i = 0; i < list.size(); i++) {  
        list.remove(i);  
    }  
}
```

Exercise

- Write a method `intersect` that accepts two sorted array lists of integers as parameters and returns a new list that contains only the elements that are found in both lists.
 - Example: if lists named `list1` and `list2` initially store:
[1, **4**, 8, 9, **11**, 15, 17, **28**, 41, **59**]
[**4**, 7, **11**, **17**, 19, 20, 23, **28**, 37, **59**, 81]
 - Then the call of `intersect(list1, list2)` returns the list:
[4, 11, 17, 28, 59]

Other Exercises

- Write a method `reverse` that reverses the order of the elements in an `ArrayList` of strings.
- Write a method `capitalizePlurals` that accepts an `ArrayList` of strings and replaces every word ending with an "s" with its uppercased version.
- Write a method `removePlurals` that accepts an `ArrayList` of strings and removes every word in the list ending with an "s", case-insensitively.

Objects storing collections

- An object can have an array, list, or other collection as a field.

```
public class Course {  
    private double[] grades;  
    private ArrayList<String> studentNames;  
  
    public Course() {  
        grades = new double[4];  
        studentNames = new ArrayList<String>();  
        ...  
    }  
}
```

- Now each object stores a collection of data inside it.

The compareTo method (10.2)

- The standard way for a Java class to define a comparison function for its objects is to define a `compareTo` method.
 - Example: in the `String` class, there is a method:

```
public int compareTo(String other)
```
- A call of **A**.`compareTo`(**B**) will 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.

Using compareTo

- `compareTo` can be used as a test in an `if` statement.

```
String a = "alice";  
String b = "bob";  
if (a.compareTo(b) < 0) { // true  
    ...  
}
```

Primitives	Objects
<code>if (a < b) { ...</code>	<code>if (a.compareTo(b) < 0) { ...</code>
<code>if (a <= b) { ...</code>	<code>if (a.compareTo(b) <= 0) { ...</code>
<code>if (a == b) { ...</code>	<code>if (a.compareTo(b) == 0) { ...</code>
<code>if (a != b) { ...</code>	<code>if (a.compareTo(b) != 0) { ...</code>
<code>if (a >= b) { ...</code>	<code>if (a.compareTo(b) >= 0) { ...</code>
<code>if (a > b) { ...</code>	<code>if (a.compareTo(b) > 0) { ...</code>

compareTo and collections

- You can use an array or list of strings with Java's included binary search method because it calls `compareTo` internally.

```
String[] a = {"al", "bob", "cari", "dan", "mike"};  
int index = Arrays.binarySearch(a, "dan"); // 3
```

- Java's `TreeSet/Map` use `compareTo` internally for ordering.

```
Set<String> set = new TreeSet<String>();  
for (String s : a) {  
    set.add(s);  
}  
System.out.println(s);  
// [al, bob, cari, dan, mike]
```

Ordering our own types

- We cannot binary search or make a `TreeSet/Map` of arbitrary types, 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.TreeSet.add(TreeSet.java:238)
```

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 to your `compareTo` method should return:
 - a value < 0 if the `other` object comes "before" this one,
 - a value > 0 if the `other` object comes "after" this one,
 - or 0 if the `other` object is considered "equal" to this.
- If you want multiple orderings, use a `Comparator` instead (see Ch. 13.1)

Comparable template

```
public class name implements Comparable<name> {  
  
    ...  
  
    public int compareTo(name other) {  
        ...  
    }  
}
```

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`

– NOTE: This trick doesn't work for `doubles` (but see `Math.signum`)

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());
}
```

Exercises

- Make the `HtmlTag` class from HTML Validator comparable.
 - Compare tags by their elements, alphabetically by name.
 - For the same element, opening tags come before closing tags.

```
// <body><b></b><i><b></b><br/></i></body>
Set<HtmlTag> tags = new TreeSet<HtmlTag>();
tags.add(new HtmlTag("body", true));    // <body>
tags.add(new HtmlTag("b", true));        // <b>
tags.add(new HtmlTag("b", false));       // </b>
tags.add(new HtmlTag("i", true));        // <i>
tags.add(new HtmlTag("b", true));        // <b>
tags.add(new HtmlTag("b", false));       // </b>
tags.add(new HtmlTag("br"));             // <br/>
tags.add(new HtmlTag("i", false));       // </i>
tags.add(new HtmlTag("body", false));    // </body>
System.out.println(tags);
// [<b>, </b>, <body>, </body>, <br/>, <i>, </i>]
```

Exercise solution

```
public class HtmlTag implements Comparable<HtmlTag> {
    ...
    // Compares tags by their element ("body" before "head"),
    // breaking ties with opening tags before closing tags.
    // Returns < 0 for less, 0 for equal, > 0 for greater.
    public int compareTo(HtmlTag other) {
        int compare = element.compareTo(other.getElement());
        if (compare != 0) {
            // different tags; use String's compareTo result
            return compare;
        } else {
            // same tag
            if ((isOpenTag == other.isOpenTag()) {
                return 0;    // exactly the same kind of tag
            } else if (other.isOpenTag()) {
                return 1;    // he=open, I=close; I am after
            } else {
                return -1;   // I=open, he=close; I am before
            }
        }
    }
}
```