The Comparable Interface

reading: 10.2
Binary search and objects

- Can we `binarySearch` an array of Strings?
  - Operators like `<` and `>` do not work with `String` objects.
  - But we do think of strings as having an alphabetical ordering.

- **natural ordering**: Rules governing the relative placement of all values of a given type.

- **comparison function**: Code that, when given two values `A` and `B` of a given type, decides their relative ordering:
  - `A < B`, `A == B`, `A > B`
**Collections class**

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarySearch(list, value)</td>
<td>returns the index of the given value in a sorted list (&lt; 0 if not found)</td>
</tr>
<tr>
<td>copy(listTo, listFrom)</td>
<td>copies listFrom's elements to listTo</td>
</tr>
<tr>
<td>emptyList(), emptyMap(),</td>
<td>returns a read-only collection of the given type that has no elements</td>
</tr>
<tr>
<td>emptySet()</td>
<td></td>
</tr>
<tr>
<td>fill(list, value)</td>
<td>sets every element in the list to have the given value</td>
</tr>
<tr>
<td>max(collection), min(collection)</td>
<td>returns largest/smallest element</td>
</tr>
<tr>
<td>replaceAll(list, old, new)</td>
<td>replaces an element value with another</td>
</tr>
<tr>
<td>reverse(list)</td>
<td>reverses the order of a list's elements</td>
</tr>
<tr>
<td>shuffle(list)</td>
<td>arranges elements into a random order</td>
</tr>
<tr>
<td>sort(list)</td>
<td>arranges elements into ascending order</td>
</tr>
</tbody>
</table>
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:
    ```java
    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,
    - 0 if `A` and `B` are considered "equal" in the ordering.
Using `compareTo`

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

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

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a &lt; b) { ... }</td>
<td>if (a.compareTo(b) &lt; 0) { ... }</td>
</tr>
<tr>
<td>if (a &lt;= b) { ... }</td>
<td>if (a.compareTo(b) &lt;= 0) { ... }</td>
</tr>
<tr>
<td>if (a == b) { ... }</td>
<td>if (a.compareTo(b) == 0) { ... }</td>
</tr>
<tr>
<td>if (a != b) { ... }</td>
<td>if (a.compareTo(b) != 0) { ... }</td>
</tr>
<tr>
<td>if (a &gt;= b) { ... }</td>
<td>if (a.compareTo(b) &gt;= 0) { ... }</td>
</tr>
<tr>
<td>if (a &gt; b) { ... }</td>
<td>if (a.compareTo(b) &gt; 0) { ... }</td>
</tr>
</tbody>
</table>
// Returns the index of an occurrence of target in a, 
// or a negative number if the target is not found. 
// Precondition: elements of a are in sorted order 
public static int binarySearch(String[] a, int target) {
    int min = 0;
    int max = a.length - 1;

    while (min <= max) {
        int mid = (min + max) / 2;
        if (a[mid].compareTo(target) < 0) {
            min = mid + 1;
        } else if (a[mid].compareTo(target) > 0) {
            max = mid - 1;
        } else {
            return mid; // target found
        }
    }

    return -(min + 1); // target not found
}
**compareTo and collections**

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

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

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

  ```java
  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.

```java
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)
```
Interfaces (9.5)

- **interface**: A list of methods that a class can promise to implement.
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A Lawyer can be treated as an Employee and inherits its code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A Rectangle object can be treated as a Shape but inherits no code.
  - Analogous to non-programming idea of roles or certifications:
    - "I'm certified as a CPA accountant. This assures you I know how to do taxes, audits, and consulting."
    - "I'm 'certified' as a Shape, because I implement the Shape interface. This assures you I know how to compute my area and perimeter."
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 `this` object comes "before" other one,
- a value > 0 if the `this` object comes "after" other one,
- 0 if the `this` object is considered "equal" to other.
Cat cat1 = new Cat("Clawdia", 6, 3.6);
Cat cat2 = new Cat("Shakespurr", 2, 1.4);
Cat cat3 = new Cat("Grumpy", 6, 3.8);

List<Cat> cats = Arrays.asList(cat1, cat2, cat3);
Collections.sort(cats);

- What is the order of the list after we sort?

[cat1, cat2, cat3]
[cat1, cat3, cat2]
[cat2, cat1, cat3]
[cat2, cat3, cat1]
[cat3, cat1, cat2]
[cat3, cat2, cat1]

// first sort on lives in ascending order
// then sort on cuteness in descending order
public int compareTo(Cat other) {
    if (other.lives != this.lives) {
        return this.lives - other.lives;
    } else {
        return (int) (other.cuteness - this.cuteness);
    }
}
Comparable template

```java
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:

```java
// 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 \(x > \text{other}.x\), then \(x - \text{other}.x > 0\)
  - if \(x < \text{other}.x\), then \(x - \text{other}.x < 0\)
  - if \(x == \text{other}.x\), then \(x - \text{other}.x == 0\)

- **NOTE**: This trick doesn't work for doubles (but see `Math.signum`)
*delegation trick* - If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```java
// 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:

```java
// 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.

```java
// <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
            }
        }
    }
}