Searching and sorting in the Java class libraries

Algorithms

- Java has *polymorphic* algorithms to provide functionality for different types of collections
  - Sorting (e.g. sort)
  - Shuffling (e.g. shuffle)
  - Routine Data Manipulation (e.g. reverse, addAll)
  - Searching (e.g. binarySearch)
  - Composition (e.g. frequency)
  - Finding Extreme Values (e.g. max)

Sequential search

- Searching for a particular word in an **`ArrayList`** ***words***

```
int index = words.indexOf(word);
if (index >= 0) {
    System.out.println(word + " is word #"
  + index);
} else {
    System.out.println(word + " is not
  found.");
}
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*...*
value	It	was	the	best	of	times	it	...

- **sequential search**: One that examines each element of a list in sequence until it finds the target value or reaches the end of the list.
  - The `indexOf` method above uses a sequential search.

Binary search algorithm

- **binary search**: An algorithm that searches for a value in a **sorted list** by repeatedly eliminating half the list from consideration.
  - Can be written iteratively or recursively
  - implemented in Java as method `Arrays.binarySearch` in `java.util` package

Binary search algorithm

- Algorithm pseudocode (searching for a value K):
  - Start out with the search area being from indexes 0 to *length*-1.
  - Examine the element in the middle of the search area.
    - If it is K, stop.
    - Otherwise,
      - If it is smaller than K, eliminate the upper half of the search area.
      - If it is larger than K, eliminate the lower half of the search area.
      - Repeat the above examination.

Binary search example

```
Passes of a binary search for a number between 1 and 100.

0     10    20    30    40    50    60    70    80    90    100

guess:  50
response/hint:  higher
conclusion:  51 <= number <= 100

0     10    20    30    40    50    60    70    80    90    100

guess:  75
response/hint:  lower
conclusion:  51 <= number <= 74

0     10    20    30    40    50    60    70    80    90    100

guess:  62
response/hint:  higher
conclusion:  63 <= number <= 74

0     10    20    30    40    50    60    70    80    90    100

guess:  68
response/hint:  higher
conclusion:  69 <= number <= 74

0     10    20    30    40    50    60    70    80    90    100

guess:  71
response/hint:  higher
conclusion:  72 <= number <= 74

0     10    20    30    40    50    60    70    80    90    100

guess:  73
response/hint:  correct!
```

Using binarySearch

Java provides two binary search methods:

- `Arrays.binarySearch` (for an array)

```
// binary search on an array:
int[] numbers = {-3, 2, 8, 12, 17, 29, 44, 58, 79};
int index = Arrays.binarySearch(numbers, 29);
System.out.println("29 is found at index " + index);
```

- `Collections.binarySearch (for a List)`

Using binarySearch

```
// binary search on ArrayList with the same values:
int index = Collections.binarySearch(list,29);
System.out.println("29 is found at index " +
index);
```

- Note that the values in the array / list are in sorted order.
- If they are not, binarySearch is not guaranteed to work properly.

Randomly shuffling a List

```
import java.util.*;

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

Shuffling

- **shuffling**: Rearranging the elements of a list into a random order.
  - Java has a shuffle method for a `List`, **`Collections.shuffle`**:

```
String[] ranks = {"2", "3", "4", "5", "6", "7", "8", "9",
                     "10", "Jack", "Queen", "King", "Ace"};
String[] suits = {"Clubs", "Diamonds", "Hearts", Spades"};
  ArrayList<String> deck = new ArrayList<String>();
  for (String rank : ranks) {    // build sorted deck
      for (String suit : suits) {
          deck.add(rank + " of " + suit);
      }
  }
  Collections.shuffle(deck);
  System.out.println("Top card = " + deck.get(0));
```

  - Output (for one example run):

```
Top card = 10 of Spades
```

Sorting

- **sorting**: Rearranging the values in a list into a given order (often into their natural ascending order).
  - One of the fundamental problems in computer science
  - Many sorts are *comparison-based* (must determine order through comparison operations on the input data)

    <, >, `compareTo`, …

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
value	15	2	8	1	17	10	12	5

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
value	1	2	5	8	10	12	15	17

Sorting in the class libraries

- Java provides two sorting methods:
  - `Arrays.sort` (for an array)

```
// demonstrate the Arrays.sort method
String[] strings = {"c", "b", "g", "h", "d", "f", "e",
"a"};
System.out.println(Arrays.toString(strings));
Arrays.sort(strings);
System.out.println(Arrays.toString(strings));
```

  - Output:

```
[c, b, g, h, d, f, e, a]
[a, b, c, d, e, f, g, h]
```

  - **`Collections.sort`** (for a `List`)

Custom ordering : Comparisons

Comparing objects;
`Comparable`, `compareTo`, and
`Comparator`

Comparing objects

- Operators like < and > do not work with objects in Java.
  - But we do think of some types as having an ordering (e.g. `Dates`).
  - (In other languages, we can enable <, > with *operator overloading*.)

- **natural ordering**: Rules governing the relative placement of all values of a given type.
  - Implies a notion of equality (like `equals`) but also < and > .
  - **total ordering**: All elements can be arranged in A ≤ B ≤ C ≤ ... order.

- **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

The Comparable interface

- The standard way for a Java class to define a comparison function for its objects is to implement the `Comparable` interface.

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

- 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 exactly 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
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`

compareTo example

```
public class Point implements Comparable<Point> {
// 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
        }
    }

    // subtraction trick:
    // return (x != other.x) ? (x - other.x) : (y - other.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` (`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());
}
```

`compareTo` and collections

- Java's binary search methods call `compareTo` internally.

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

- Java's `TreeSet`/`TreeMap` use `compareTo` internally for ordering.
  - Only classes that implement `Comparable` can be used as elements.

```
Set<String> set = new TreeSet<String>();
for (int i = a.length - 1; i >= 0; i--) {
    set.add(a[i]);
}
System.out.println(s);
// [al, bob, cari, dan, mike]
```

Flawed compareTo method

```
public class BankAccount implements
  Comparable<BankAccount> {
    private String name;
    private double balance;
    private int id;
    ...
    public int compareTo(BankAccount other) {
        return name.compareTo(other.name);  // order by
  name
    }

    public boolean equals(Object o) {
        if (o != null && getClass() == o.getClass()) {
            BankAccount ba = (BankAccount) o;
            return name.equals(ba.name)
                && balance == ba.balance && id == ba.id;
        } else {
            return false;
        }
    }
}
```

- What's bad about the above?

The flaw

```
BankAccount ba1 = new BankAccount("Jim", 123, 20.00);
BankAccount ba2 = new BankAccount("Jim", 456, 984.00);

Set<BankAccount> accounts = new TreeSet<BankAccount>();
accounts.add(ba1);
accounts.add(ba2);
System.out.println(accounts);     // [Jim($20.00)]
```

- Where did the other account go?
  - Since the two accounts are "equal" by the ordering of `compareTo`, the set thought they were duplicates and didn't store the second.

compareTo and equals

- `compareTo` should generally be consistent with `equals`.
  - `a.compareTo(b) == 0` should imply that `a.equals(b)`.

- from `Comparable` Java API docs:
  - ... sorted sets (and sorted maps) without explicit comparators behave strangely when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals method.
  - For example, if one adds two keys a and b such that (!a.equals(b) && a.compareTo(b) == 0) to a sorted set that does not use an explicit comparator, the second add operation returns false (and the size of the sorted set does not increase) because a and b are equivalent from the sorted set's perspective.

What's the "natural" order?

```
public class Rectangle implements Comparable<Rectangle> {
    private int x, y, width, height;

    public int compareTo(Rectangle other) {
        // ...?
    }
}
```

- What is the "natural ordering" of rectangles?
  - By x, breaking ties by y?
  - By width, breaking ties by height?
  - By area? By perimeter?

- Do rectangles have any "natural" ordering?
  - Might we ever want to sort rectangles into some order anyway?

Comparator interface

```
public interface Comparator<T> {
    public int compare(T first, T second);
}
```

- Interface `Comparator` is an external object that specifies a comparison function over some other type of objects.
  - Allows you to define multiple orderings for the same type.
  - Allows you to define a specific ordering for a type even if there is no obvious "natural" ordering for that type.

Comparator examples

```
public class RectangleAreaComparator
        implements Comparator<Rectangle> {
    // compare in ascending order by area (WxH)
    public int compare(Rectangle r1, Rectangle
  r2) {
        return r1.getArea() - r2.getArea();
    }
}

public class RectangleXYComparator
        implements Comparator<Rectangle> {
    // compare by ascending x, break ties by y
    public int compare(Rectangle r1, Rectangle
  r2) {
        if (r1.getX() != r2.getX()) {
            return r1.getX() - r2.getX();
        } else {
            return r1.getY() - r2.getY();
        }
    }
}
```

Using Comparators

- `TreeSet` and `TreeMap` can accept a `Comparator` parameter.

```
Comparator<Rectangle> comp = new
  RectangleAreaComparator();
Set<Rectangle> set = new
  TreeSet<Rectangle>(comp);
```

- Searching and sorting methods can accept `Comparators`.

`Arrays.binarySearch(`**array**`, `**value**`, `**comparator**`)`
`Arrays.sort(`**array**`, `**comparator**`)`
`Collections.binarySearch(`**list**`, `**comparator**`)`
`Collections.max(`**collection**`, `**comparator**`)`
`Collections.min(`**collection**`, `**comparator**`)`
`Collections.sort(`**list**`, `**comparator**`)`

- Methods are provided to reverse a `Comparator`'s ordering:

`Collections.reverseOrder()`
`Collections.reverseOrder(`**comparator**`)`

Using a separate Comparator

- Program implemented Comparable
  - Therefore, it had a compareTo method
  - We could sort *only* by their score
  - If we wanted to sort students another way, such as by name, we are out of luck
- Now we will put the comparison method in a *separate class* that implements Comparator instead of Comparable
  - This is more flexible (you can use a different Comparator to sort Students by name or by score), but it's also clumsier
  - Comparator is in java.util, not java.lang
  - Comparable requires a definition of compareTo but Comparator requires a definition of compare
  - Comparator also (sort of) requires equals

Outline of StudentComparator

```
import java.util.*;

public class StudentComparator
                implements Comparator {

    public int compare(Object o1, Object o2) {...}

    public boolean equals(Object o1) {...}
}
```

- Note: When we are using this Comparator, we don't need the `compareTo` method in the `Student` class

The compare method

```
public int compare(Object o1, Object o2) {
        return ((Student)o1).score - ((Student)o2).score;
}
```

- This differs from `compareTo(Object o)` in `Comparable` in these ways:
  - The name is different
  - It takes both objects as parameters, not just one
  - We have to check the type of both objects
  - Both objects have to be cast to `Student`

Custom ordering

- Sometimes, the default ordering is not what you want.
  - Example: The following code sorts the strings in a case-sensitive order, so the uppercase letters come first. We may have wanted a case-insensitive ordering instead.

```
String[] strings = {"Foxtrot", "alpha", "echo",
"golf‶,"bravo", "hotel", "Charlie", "DELTA"};
Arrays.sort(strings);
System.out.println(Arrays.toString(strings));
```

  - Output:

```
[Charlie, DELTA, Foxtrot, alpha, bravo, echo, golf,
hotel]
```

- You can describe a custom sort ordering by creating a class called a *comparator*.

Comparator example

- The following `Comparator` compares `String`s, ignoring case:

```
public class CaseInsensitiveComparator
        implements Comparator<String> {

 public int compare(String s1, String s2) {
        return s1.toLowerCase().compareTo(
                s2.toLowerCase());
    }
}
```

Sorting with Comparators

- The sorting methods shown previously can also be called with a Comparator as a second parameter.
  - The sorting algorithm will use that comparator to order the elements of the array or list.

```
String[] strings = {"Foxtrot", "alpha", "echo", "golf",
                    "bravo", "hotel", "Charlie", "DELTA"};
Arrays.sort(strings, new CaseInsensitiveComparator());
System.out.println(Arrays.toString(strings));
[Charlie, DELTA, Foxtrot, alpha, bravo, echo, golf, hotel]
```

  - Output:
    [alpha, bravo, Charlie, DELTA, echo, Foxtrot, golf, hotel]

The `main` method

- The `main` method is just like before, except that instead of

```
TreeSet set = new TreeSet();
```

  We have

```
Comparator comp = new StudentComparator();
TreeSet set = new TreeSet(comp);
```

When to use each

- The Comparable interface is simpler and less work
  - Your class implements Comparable
  - Provide a public int compareTo(Object o) method
  - Use no argument in your TreeSet or TreeMap constructor
  - You will use the same comparison method every time
- The Comparator interface is more flexible but slightly more work
  - Create as many different classes that implement Comparator as you like
  - You can sort the TreeSet or TreeMap differently with each
    - Construct TreeSet or TreeMap using the comparator you want
  - For example, sort Students by score *or* by name

Sorting differently

- Suppose you have students sorted by *score*, in a `TreeSet` you call `studentsByScore`
- Now you want to sort them again, this time by *name*

```
Comparator myStudentNameComparator =
        new MyStudentNameComparator();

TreeSet studentsByName =
      new TreeSet(myStudentNameComparator);

studentsByName.addAll(studentsByScore);
```

Algorithms

- See the Java API for the Collections class
  - play with max, min, sort, binarySearch, shuffle, reverse