The Java™ Tutorials

Trail: Collections

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Lesson: Algorithms

The *polymorphic algorithms* described here are pieces of reusable functionality provided by the Java platform. All of them come from the `Collections` class, and all take the form of static methods whose first argument is the collection on which the operation is to be performed. The great majority of the algorithms provided by the Java platform operate on `List` instances, but a few of them operate on arbitrary `Collection` instances. This section briefly describes the following algorithms:

- Sorting
- Shuffling
- Routine Data Manipulation
- Searching
- Composition
- Finding Extreme Values

Sorting

The `sort` algorithm reorders a `List` so that its elements are in ascending order according to an ordering relationship. Two forms of the operation are provided. The simple form takes a `List` and sorts it according to its elements' *natural ordering*. If you're unfamiliar with the concept of natural ordering, read the Object Ordering section.

The `sort` operation uses a slightly optimized *merge sort* algorithm that is fast and stable:

- **Fast**: It is guaranteed to run in `n log(n)` time and runs substantially faster on nearly sorted lists. Empirical tests showed it to be as fast as a highly optimized quicksort. A quicksort is generally considered to be faster than a merge sort but isn't stable and doesn't guarantee `n log(n)` performance.
- **Stable**: It doesn't reorder equal elements. This is important if you sort the same list repeatedly on different attributes. If a user of a mail program sorts the inbox by mailing date and then sorts it by sender, the user naturally expects that the now-contiguous list of messages from a given sender will (still) be sorted by mailing date. This is guaranteed only if the second sort was stable.

The following `trivial program` prints out its arguments in lexicographic (alphabetical) order.

```
import java.util.*;

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

Let's run the program.

```
% java Sort i walk the line
```

The following output is produced.

```
[i, line, the, walk]
```

The program was included only to show you that algorithms really are as easy to use as they appear to be.

The second form of `sort` takes a `Comparator` in addition to a `List` and sorts the elements with the `Comparator`. Suppose you want to print out the anagram groups from our earlier example in reverse order of size — largest anagram group first. The example that follows shows you how to achieve this with the help of the second form of the `sort` method.

Recall that the anagram groups are stored as values in a `Map`, in the form of `List` instances. The revised printing code iterates through the `Map`'s values view, putting every `List` that passes the minimum-size test into a `List` of `List`s. Then the code sorts this `List`, using a `Comparator` that expects `List` instances, and implements reverse size-ordering. Finally, the code iterates through the sorted `List`, printing its elements (the anagram groups). The following code replaces the printing code at the end of the `main` method in the `Anagrams` example.

```
// Make a List of all anagram groups above size threshold.
List<List<String>> winners = new ArrayList<List<String>>();
for (List<String> l : m.values())
    if (l.size() >= minGroupSize)
        winners.add(l);

// Sort anagram groups according to size
Collections.sort(winners, new Comparator<List<String>>() {
    public int compare(List<String> o1, List<String> o2) {
        return o2.size() - o1.size();
    }});

// Print anagram groups.
for (List<String> l : winners)
    System.out.println(l.size() + ": " + l);
```

Running `the program` on the `same dictionary` as in The Map Interface section, with the same minimum anagram group size (eight), produces the following output.

```
12: [apers, apres, asper, pares, parse, pears, prase,
       presa, rapes, reaps, spare, spear]
11: [alerts, alters, artels, estral, laster, ratels,
       salter, slater, staler, stelar, talers]
10: [least, setal, slate, stale, steal, stela, taels,
       tales, teals, tesla]
9: [estrin, inerts, insert, inters, niters, nitres,
       sinter, triens, trines]
9: [capers, crapes, escarp, pacers, parsec, recaps,
       scrape, secpar, spacer]
9: [palest, palets, pastel, petals, plates, pleats,
       septal, staple, tepals]
9: [anestri, antsier, nastier, ratines, retains, retinas,
       retsina, stainer, stearin]
8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]
8: [aspers, parses, passer, prases, repass, spares,
       sparse, spears]
8: [enters, nester, renest, rentes, resent, tenser,
       ternes,��treens]
8: [arles, earls, lares, laser, lears, rales, reals, seral]
8: [earings, erasing, gainers, reagins, regains, reginas,
       searing, seringa]
8: [peris, piers, pries, prise, ripes, speir, spier, spire]
8: [ates, east, eats, etas, sate, seat, seta, teas]
8: [carets, cartes, caster, caters, crates, reacts,
       recast,��traces]
```

Shuffling

The `shuffle` algorithm does the opposite of what `sort` does, destroying any trace of order that may have been present in a `List`. That is, this algorithm reorders the `List` based on input from a source of randomness such that all possible permutations occur with equal likelihood, assuming a fair source of randomness. This algorithm is useful in implementing games of chance. For example, it could be used to shuffle a `List` of `Card` objects representing a deck. Also, it's useful for generating test cases.

This operation has two forms: one takes a `List` and uses a default source of randomness, and the other requires the caller to provide a Random object to use as a source of randomness. The code for this algorithm is used as an example in the `List` section.

Routine Data Manipulation

The `Collections` class provides five algorithms for doing routine data manipulation on `List` objects, all of which are pretty straightforward:

- `reverse` — reverses the order of the elements in a `List`.
- `fill` — overwrites every element in a `List` with the specified value. This operation is useful for reinitializing a `List`.
- `copy` — takes two arguments, a destination `List` and a source `List`, and copies the elements of the source into the destination, overwriting its contents. The destination `List` must be at least as long as the source. If it is longer, the remaining elements in the destination `List` are unaffected.
- `swap` — swaps the elements at the specified positions in a `List`.
- `addAll` — adds all the specified elements to a `Collection`. The elements to be added may be specified individually or as an array.

Searching

The `binarySearch` algorithm searches for a specified element in a sorted `List`. This algorithm has two forms. The first takes a `List` and an element to search for (the "search key"). This form assumes that the `List` is sorted in ascending order according to the natural ordering of its elements. The second form takes a `Comparator` in addition to the `List` and the search key, and assumes that the `List` is sorted into ascending order according to the specified `Comparator`. The `sort` algorithm can be used to sort the `List` prior to calling `binarySearch`.

The return value is the same for both forms. If the `List` contains the search key, its index is returned. If not, the return value is `(-(insertion point) - 1)`, where the insertion point is the point at which the value would be inserted into the `List`, or the index of the first element greater than the value or `list.size()` if all elements in the `List` are less than the specified value. This admittedly ugly formula guarantees that the return value will be `>= 0` if and only if the search key is found. It's basically a hack to combine a boolean `(found)` and an integer `(index)` into a single `int` return value.

The following idiom, usable with both forms of the `binarySearch` operation, looks for the specified search key and inserts it at the appropriate position if it's not already present.

```
int pos = Collections.binarySearch(list, key);
if (pos < 0)
   l.add(-pos-1, key);
```

Composition

The frequency and disjoint algorithms test some aspect of the composition of one or more `Collections`:

- `frequency` — counts the number of times the specified element occurs in the specified collection
- `disjoint` — determines whether two `Collections` are disjoint; that is, whether they contain no elements in common

Finding Extreme Values

The `min` and the `max` algorithms return, respectively, the minimum and maximum element contained in a specified `Collection`. Both of these operations come in two forms. The simple form takes only a `Collection` and returns the minimum (or maximum) element according to the elements' natural ordering. The second form takes a `Comparator` in addition to the `Collection` and returns the minimum (or maximum) element according to the specified `Comparator`.

Previous page: Previous Lesson
Next page: Custom Collection Implementations