Contents

0 Introduction

So far, we have had the "insider's view" of data structures and have gained insight into how they work by doing some building from scratch. Now it is time for us to explore Java's Collection Framework, a set of classes that will enable you to accomplish complex data manipulation quickly and efficiently, as well as some useful interfaces that will allow you to do custom sorts.

To fully take advantage of power of the Collections Framework, we need to spend a little time talking about the issues of ordering and sorting. Sorting is done according to some kind of ordering, and we will begin with a brief mathematical disquisition on orderings.

Once we discuss this, we will look at the Comparable and Comparator interfaces. We will use thesde in conjunction with the static service class `Collections` to do some custom sorting.

Finally, we will turn our attention to Java's two frameworks for collections, the Collections and Map Frameworks, and create some sample applications based on these.

1 A Mathematical Overture

What we are going to do here is to introduce some useful mathematical abstractions that arise from sorting and comparing objects. Being aware of these abstractions will tune you into the aesthetical considerations that go into the design contracts for the `Comparable` and `Comparator` interfaces. It also will provide a succinct language for describing what is happening in the contract.

Suppose that A and B are sets. We define the *cartesian product* of A and B by

$$A \times B = \{(a, b) | a \in A, b \in B\}.$$

Simply put, this is the set of all ordered pairs with the first element chosen from A and the second element chosen from B. Two ordered pairs are equal if their

first and second elements are equal. To wit,

$$(a_0, b_0) = (a_1, b_1) \iff (a_0 = a_1) \wedge (b_0 = b_1).$$

We could construct a class for this in Java as follows.

```
public class Pair<S, T>
{
    private S first;
    private T second;
    public Pair(S first, T second)
    {
        this.first = first;
        this.second = second;
    }
    @Override
    @SuppressWarnings("unchecked")
    public boolean equals(Object o)
    {
        if(! (o instanceof Pair))
        {
            return false;
        }
        Pair<S, T> that = (Pair<S, T>) o;
        return first.equals(that.first)
            && second.equals(that.second);
    }
    @Override
    public String toString()
    {
        return "S: " + first.toString() + ", T: "
            + second.toString();
    }
    public static void main(String[] args)
    {
        Pair<String, String> terminator
            = new Pair<String, String>("Arnold", "Schwarznegger");
        Pair<Integer, Integer> point = new Pair<Integer, Integer>(3,5);
        Pair<Integer, Integer> anotherPoint
            = new Pair<Integer, Integer>(3,5);
        if(point.equals(anotherPoint))
        {
            System.out.printf("%s equals %s\n", point, anotherPoint);
        }
        else
        {
```

```
            System.out.printf("%s does not equal %s\n",
                point, anotherPoint);
        }
        System.out.println(terminator);
    }
}
```

You should play with this class by inserting more examples into its main method and running it.

Now if X is a set, we define a *relation* on X to be a subset of $X \times X$; i.e. a relation is a set of ordered pairs of elements of X. This may seems a strange construct, but let us bring it down to earth.

If X is any set put $R = \{(x, x) | x \in X\}$. You can think of this as being the "main diagonal" in $X \times X$. Suppose that $(x, y) \in R$ Then $(x, y) = (a, a)$ for some $a \in X$. but then $a = x = y$, so $x = y$. Conversely, if $x = y$, then $(x, y) \in R$. This is, in fact the relation of equality on any set. The "diagonal relation" is equality. You can think of this as an equality of identity check for objects.

It is common to use infix notation to describe relations. If R is a relation, we will often write $x\,R\,y$ if $(x, y) \in R$.

There are several properties relations can have. These include

- **reflexivity** A relation on a set X is reflexive if for all $x \in X$, $x\,R\,x$, i.e, every element is related to itself.
- **symmetry** A relation on a set X is symmetric if for any $x, y \in X$, if $x\,R\,y$ then $y\,R\,x$.
- **antisymmetry** A relation on a set X is antisymmetric if for any $x, y \in X$, if $x\,R\,y$ and $y\,R\,X$ occurs only if $y = x$.
- **transitive** A relation R on a set X is transitive if for any $x, y, z \in X$, $(x\,R\,y) \vee (y\,R\,z) \Rightarrow (x\,R\,z)$.
- An *equivalence relation* is a relation that is symmetric, transitive and symmetric.
- A *nonstrict partial order* is a reflexive, antisymmetric, transitive relation on a set.
- A *strict partial order* is a an antisymmetric and transitive relation on a set.
- A *linear* or *total order* is a relation R on a set X so that for any $x, y \in X$, we have $(x\,R\,y) \vee (y\,R\,x)$.

You might ask, "How does this relate to such familiar stuff as $\geq$ we encounter in real and integer arithemetic? Let us give a simple example.

The cartesian plane we all know and love from analytic geometry is just $\mathbb{R} \times \mathbb{R}$. Let us define the subset R to be the set of all points on the line $y = x$ along with those above it. This is a "half-plane".

Now suppose $(x, y) \in R$. Then the point (x, y) lies on or above the line $y = x$; this means that $y \geq x$. Conversely, if $y \geq x$ then (x, y) lies above the line $y = x$. So, the relation R, satisfyingly, is just familiar old $\geq$.

Matematical Exercises

1. What subset of the plane represents $\leq$? Draw it and execute an argument similar to the one above to show you are correct.
2. What subset of the plane represents $<$?
3. What subset of the plane reprsents the relation $\neq$?
4. What relation properties are had by $\leq$? Is it a partial order? A strict partial order?
5. Suppose that $\mathcal{S}$ is the set of all character strings. What kind of relation is asciicographical order?
6. Let us define for $x, y \in \mathcal{C}$, $x\, R\, y$ if x`.toLowerCase().equals(`y`.toLowerCase()` evaluates to `true`. What kind of relation is this?

2 `Comparable`, `Comparator`, `Coollections`, and Sorting

The package `java.util` contains the interface `Iterator<E>`, which specifies three methods `public E next()`, `public boolean hasNext()`, and `public E remove()`. We used this interface in our stack and list classes. Iterator are used to traverse containers.

The package `java.lang` has these three interfaces. All of these specify exactly one method, so they are functional interfaces.

- `Iterable<E>`, which specifies `public Iterator<E> iterator()` This enables the collections `for` loop.
- `Comparable<E>`, which specifies `public int compareTo(E other)`
- `Comparator<E>`, which specifies `public int compare(E x, E y)`

The design contract for `Comparable<E>` and `Comparator<E>` calls for their specified methods to impose an order on items of type `E` that is an linear ordering.

The design contract for the `equals` method we have impleamented several times is that is is an equivalence relation.

Sorting is done using a `sort key`; this is a numerical function defined on objects that imposes an order. Java implements these via the `Comparable` and `Comparator` intefaces.

A sort key you have seen already is `compareTo` on strings. You know that if `s` and `t` are strings, that `s` precedes `t` alphabetically precisely when `a.compareToIgnoreCase(b) < 0`.

2.1 The Static Sevice Class `Collections`

This class has a host of useful methods, all of which are static. We will pay attention to just a few of them for now.

- `shuffle(List<?>`, which shuffles a list in-place; this works very nicely on array lists. It uses the Fisher-Yates procedure for shuffling elements.
- `sort(List<T> list)`, which will sort a list in-place provided that `T` is a subtype of `Comparable`.
- `sort(List<T>, Comparator<? super T>)`, which sorts a list in-place using a `Comparator` as a sort key.

Sorts in Java are done using a method called *timsort*, which is an exampe of a *stable sort*, i.e., items with equal keys do not change order during the sort.

Let us do a simple example in `jshell`. First we make a comparator.

```
jshell> Comparator<String> alpha = (a, b) -> a.compareToIgnoreCase(b);
alpha ==> $Lambda$13/703504298@c038203
```

Now let us make an array list and populate it.

```
jshell> ArrayList<String> al = new ArrayList<String>();
al ==> []

jshell> al.addAll(Arrays.asList("zither", "Aardvark", "Anderson", "blueberry", "daffodil", '
$3 ==> true
jshell> al
al ==> [zither, Aardvark, Anderson, blueberry, daffodil, HELLO]
```

Now we will use the method `Collections.sort`; the result is an asciicographical sort of or strings.

```
jshell> Collections.sort(al);
jshell> al
al ==> [Aardvark, Anderson, HELLO, blueberry, daffodil, zither]
```

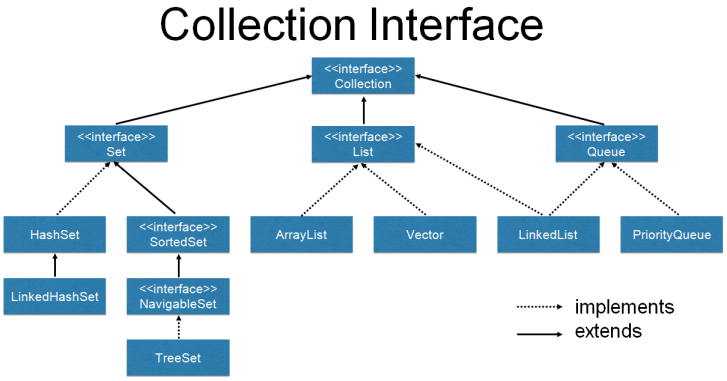
Now we will use the method `Collections.sort` that allows you to pass in a comparator. We now get an alphabetical, case-insensitive sort.

```
jshell> Collections.sort(al, alpha);

jshell> al
al ==> [Aardvark, Anderson, blueberry, daffodil, HELLO, zither]
```

In this chapter, we will begin by learning about the interface `Collection` and the static service class `Collections`

The interface `Collection` is a subinterface of `Iterable`. You will see that `ArrayList` implements this interface. This interface specifies what a basic Java Standard Library generic container class shoud look like.



An important service provided by the class `Collections` is that of sorting. How do we control how items in a collection get sorted? What constitutes a sortable item? We will answer these questions and apply them to some probems. The `Comparable` and `Comparator` interfaces will help us to deal with sortable collections; these interfaces enjoy the advantage of being fuctional interfaces.

Along the way, we shall learn how the static service class `Collections` allows us to control sorting using the `Comparable` and `Comparator` interfaces.