COP 3330 – Object Oriented Programming in Java
Java Class: HashSet

HashSet

A set in mathematics is a collection of objects without regard to repeats. All valid English words comprise a set, for example. A very common task in maintaining databases is to maintain a set of objects dynamically, where an object can be added to the set, an object can be removed from the set, and we can check to see if an object is in the set.

A very simple way to implement this functionality is in an ArrayList using for loops. This works, but it's quite slow. Each operation could take O(n) time, where n is the number of items in the list.

If we knew that all of the items in our set had to be integers in between 0 and n – 1, we could keep a boolean array of size n, and in each index store true if the index value was in the set, and false otherwise. But if the items we are storing are of an arbitrary type, this solution won't work.

In COP 3502, students learn how to use hash functions and hash tables to provide O(1) time on average, for each of these operations.

For convenience, Java provides this functionality (and speed) without its users having to understand the nuances of hash functions. Just like ArrayList, HashSet is a collection and uses generics.

We typically use the default HashSet constructor. Here is a list of the important HashSet methods:

```
// Attempts to add e to this HashSet and returns true if this
// set didn't previously contain e.
boolean add(E e)

// Clears the contents of this HashSet
void clear()

// Returns true if and only if this HashSet contains o.
boolean contains(Object o)

// Attempts to remove e from this HashSet. Returns true if the
// element e was previously in this HashSet.
boolean remove(E e)

// Returns the number of elements in this HashSet.
int size()
```

Grocery Example

Here is a short hard-coded example of a HashSet of Strings storing groceries to illustrate the use of each of these methods. One of the things you'll note is that an iterator style loop is required to loop through each element in a HashSet.

```
import java.util.*;

public class grocery {

    public static void main(String[] args) {

        HashSet<String> groceries = new HashSet<String>();
        groceries.add("apple");
        groceries.add("banana");
        groceries.add("apple");
        groceries.add("kiwi");
        printSet(groceries);

        groceries.remove("apple");
        printSet(groceries);

        String[] stuff = {"dragonfruit", "kiwi", "apple", "banana",
                          "strawberry"};

        for (String x : stuff) {
            if (groceries.contains(x))
                System.out.println("Yes, we have "+x+".");
            else
                System.out.println("No, we don't have "+x+".");
        }

        groceries.clear();
        printSet(groceries);

        for (String x: stuff)
            groceries.add(x);
        for (String x: stuff)
            groceries.add(x);
        printSet(groceries);

        System.out.println("We have "+groceries.size()+" grocery items.");
    }

    public static void printSet(HashSet<String> set) {
        System.out.print("Set: ");
        for (String s: set)
            System.out.print(s+" ");
        System.out.println();
    }
}
```

In the beginning of the example we add several items, including one repeat, to the set. Then, we call the printSet method. Notice the key technique to loop through all items in a set via the iterator loop:

```
for (String s: set) {
    // use s in here
}
```

when we write code like this, the loop is guaranteed to go through each item that belongs to set and within the loop body the way to refer to the current item is the reference s.

We can similarly use this syntax with an array (which occurs later in the example), but are not required to.

This first print prints out:

```
Set: banana apple kiwi
```

Then we test the remove method. Even though we added apple twice, when we remove it once, it's gone (because there were never two apples in the set to begin with.) The second print is:

```
Set: banana kiwi
```

Before we move further, note that the items in a HashSet are unordered, so they are liable to print out in any random order and there's no guarantee for what order an iterator loop will run. (Thus, we must write our code in such a way that the order that we iterate through the set items doesn't matter.)

We test out the contains method 5 times, getting the following output, as expected:

```
No, we don't have dragonfruit.
Yes, we have kiwi.
No, we don't have apple.
Yes, we have banana.
No, we don't have strawberry.
```

After we clear the set and print, it proves that the clear method removed all items that were previously in the set from the set.

Finally to test the size method, we add back each element from the Array (twice), and then print out the size of the set (which is 5).

Kattis Problem: CD and use of BufferedReader

Let's solve one more problem with a HashSet. We'll look at a couple of different approaches to solve it. Here is the text of the problem from Kattis: https://open.kattis.com/problems/cd. Our first approach is as follows:

1. Create a set and add all of Jack's CDs.
2. When reading through Jill's CDs determine how many are already contained in Jack's set.

Unfortunately, if we code this up by reading input with a Scanner, we'll get a time limit exceeded verdict. Scanner objects are slow because they check several things while reading in input. The BufferedReader object can also be used to read in information, but runs faster. If we know the type of the input values, then we can increase the speed of our program by using BufferedReader instead of Scanner. Unfortunately, BufferedReader reader only has a readLine() method which returns a whole like. We must use the StringTokenizer to parse out lines with more than one token on them.

If we use a BufferedReader, we must import java.io.*; because BufferedReader is in the java.io package, not java.util. We are required to report that the use of this class could throw an Exception. Here is our first CD solution:

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

public class cd1 {

    public static void main(String[] args) throws Exception {
        BufferedReader stdin = new BufferedReader(new
                               InputStreamReader(System.in));
        StringTokenizer tok  = new StringTokenizer(stdin.readLine());
        int numJack = Integer.parseInt(tok.nextToken());
        int numJill = Integer.parseInt(tok.nextToken());

        while (numJack != 0 && numJill != 0) {

            HashSet<Integer> jackCDs = new HashSet<Integer>();
            for (int i=0; i<numJack; i++) {
                int tmp = Integer.parseInt(stdin.readLine());
                jackCDs.add(tmp);
            }

            int res = 0;
            for (int i=0; i<numJill; i++) {
                int tmp = Integer.parseInt(stdin.readLine());
                if (jackCDs.contains(tmp) res++;
            }

            System.out.println(res);
            tok  = new StringTokenizer(stdin.readLine());
            numJack = Integer.parseInt(tok.nextToken());
            numJill = Integer.parseInt(tok.nextToken());
        }
    }
}
```

The second import is because we want to use BufferedReader. In addition, the "throws Exception" in the main method definition is required. It goes exactly after the parameter list for main. Finally, to create a BufferedReader to read from standard input (keyboard), you must use create an InputStreamReader objects from System.in, and use that as a parameter to the BufferedReader constructor.

The while loop is just to deal with the fact that multiple input cases are given to us in a bit of a strange way (sentinel values).

Now that we've gotten past all of the formatting, we see that the solution is quite straightforward. We create a set called jackCDs. We add to that set each of Jack's CDs. Then, we read through Jill's CDs, and for each one, we check if it's in the set jackCDs. Each item that is will add 1 to our result.

Another approach to solve this problem is to add ALL the numbers to the same set, and then use the Inclusion-Exclusion principle and the set size. If Jack has 3 CDs and Jill has 4 CDs, and together, then own 5 unique CDs, then it must be the case that 3 + 4 – 5 = 2 of the CDs are repeats, since only the repeated CDs are counted twice when adding 3 and 4. Here is that solution:

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

public class cd2 {

    public static void main(String[] args) throws Exception {

        BufferedReader stdin = new BufferedReader(new
                               InputStreamReader(System.in));
        StringTokenizer tok  = new StringTokenizer(stdin.readLine());
        int numJack = Integer.parseInt(tok.nextToken());
        int numJill = Integer.parseInt(tok.nextToken());

        while (numJack != 0 && numJill != 0) {

            HashSet<Integer> allCDs = new HashSet<Integer>();

             for (int i=0; i<numJack+numJill; i++) {
                 int tmp = Integer.parseInt(stdin.readLine());
                 allCDs.add(tmp);
             }

             // Using I/E Principle here!
             System.out.println(numJack+numJill-allCDs.size());

             tok  = new StringTokenizer(stdin.readLine());
             numJack = Integer.parseInt(tok.nextToken());
             numJill = Integer.parseInt(tok.nextToken());
        }
    }
}
```

This solution is even more compact, since we use just one for loop to process the CDs! This loop runs longer (numJack+numJill), and then we do the math in the ouput statement.