**Programming Exercises for Chapter 10**

THIS PAGE CONTAINS several exercises for Chapter 10 in [Introduction to Programming Using Java](http://math.hws.edu/eck/cs124/javanotes7/index.html). For each exercise, a link to a possible solution is provided. Each solution includes a discussion of how a programmer might approach the problem and interesting points raised by the problem or its solution, as well as complete source code of the solution.

**Exercise 10.1:**

Rewrite the *PhoneDirectory* class from [Subsection 7.4.2](http://math.hws.edu/eck/cs124/javanotes7/c7/s4.html#arrays.4.2) so that it uses a *TreeMap* to store directory entries, instead of an array. (Doing this was suggested in [Subsection 10.3.1](http://math.hws.edu/eck/cs124/javanotes7/c10/s3.html#generics.3.1).)

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c10/ex1-ans.html)

**Exercise 10.2:**

In mathematics, several operations are defined on sets. The union of two sets A and B is a set that contains all the elements that are in A together with all the elements that are in B. The intersectionof A and B is the set that contains elements that are in both A and B. The difference of A and B is the set that contains all the elements of A **except** for those elements that are also in B.

Suppose that A and B are variables of type set in Java. The mathematical operations on A and B can be computed using methods from the *Set* interface. In particular: A.addAll(B) computes the *union*of A and B; A.retainAll(B) computes the *intersection* of A and B; and A.removeAll(B) computes the *difference* of A and B. (These operations change the contents of the set A, while the mathematical operations create a new set without changing A, but that difference is not relevant to this exercise.)

For this exercise, you should write a program that can be used as a "set calculator" for simple operations on sets of non-negative integers. (Negative integers are not allowed.) A set of such integers will be represented as a list of integers, separated by commas and, optionally, spaces and enclosed in square brackets. For example: [1,2,3] or [17, 42, 9, 53, 108]. The characters +, \*, and - will be used for the union, intersection, and difference operations. The user of the program will type in lines of input containing two sets, separated by an operator. The program should perform the operation and print the resulting set. Here are some examples:

Input Output

------------------------- -------------------

[1, 2, 3] + [3, 5, 7] [1, 2, 3, 5, 7]

[10,9,8,7] \* [2,4,6,8] [8]

[ 5, 10, 15, 20 ] - [ 0, 10, 20 ] [5, 15]

To represent sets of non-negative integers, use sets of type *TreeSet<Integer>*. Read the user's input, create two *TreeSets*, and use the appropriate *TreeSet* method to perform the requested operation on the two sets. Your program should be able to read and process any number of lines of input. If a line contains a syntax error, your program should not crash. It should report the error and move on to the next line of input. (Note: To print out a *Set*, A, of *Integers*, you can just say System.out.println(A). I've chosen the syntax for sets to be the same as that used by the system for outputting a set.)

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c10/ex2-ans.html)

**Exercise 10.3:**

The fact that Java has a *HashMap* class means that no Java programmer has to write an implementation of hash tables from scratch -- unless, of course, that programmer is a computer science student.

For this exercise, you should write a hash table in which both the keys and the values are of type *String*. (This is not an exercise in generic programming; do not try to write a generic class.) Write an implementation of hash tables from scratch. Define the following methods: get(key), put(key,value), remove(key), containsKey(key), and size(). Remember that every object, obj, has a method obj.hashCode() that can be used for computing a hash code for the object, so at least you don't have to define your own hash function. Do not use **any** of Java's built-in generic types; create your own linked lists using nodes as covered in [Subsection 9.2.2](http://math.hws.edu/eck/cs124/javanotes7/c9/s2.html#recursion.2.2). However, you do **not** have to worry about increasing the size of the table when it becomes too full.

You should also write a short program to test your solution.

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c10/ex3-ans.html)

**Exercise 10.4:**

A predicate is a boolean-valued function with one parameter. Some languages use predicates in generic programming. Java 7 doesn't, but this exercise looks at how predicates might work.

In Java, we could implement "predicate objects" by defining a generic interface:

public interface Predicate<T> {

public boolean test( T obj );

}

The idea is that an object that implements this interface knows how to "test" objects of type *T* in some way. Define a class that contains the following generic static methods for working with predicate objects. The name of the class should be *Predicates*, in analogy with the standard class *Collections* that provides various static methods for working with collections.

public static <T> void remove(Collection<T> coll, Predicate<T> pred)

// Remove every object, obj, from coll for which

// pred.test(obj) is true.

public static <T> void retain(Collection<T> coll, Predicate<T> pred)

// Remove every object, obj, from coll for which

// pred.test(obj) is false. (That is, retain the

// objects for which the predicate is true.)

public static <T> List<T> collect(Collection<T> coll, Predicate<T> pred)

// Return a List that contains all the objects, obj,

// from the collection, coll, such that pred.test(obj)

// is true.

public static <T> int find(ArrayList<T> list, Predicate<T> pred)

// Return the index of the first item in list

// for which the predicate is true, if any.

// If there is no such item, return -1.

(In C++, methods similar to these are included as a standard part of the generic programming framework. And Java 8 has a similar predicate interface, and the *Collections* class has a removeIf()method that uses a predicate.)

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c10/ex4-ans.html)

**Exercise 10.5:**

An example in [Subsection 10.4.2](http://math.hws.edu/eck/cs124/javanotes7/c10/s4.html#generics.4.2) concerns the problem of making an index for a book. A related problem is making a concordance for a document. A concordance lists every word that occurs in the document, and for each word it gives the line number of every line in the document where the word occurs. All the subroutines for creating an index that were presented in [Subsection 10.4.2](http://math.hws.edu/eck/cs124/javanotes7/c10/s4.html#generics.4.2) can also be used to create a concordance. The only real difference is that the integers in a concordance are line numbers rather than page numbers.

Write a program that can create a concordance. The document should be read from an input file, and the concordance data should be written to an output file. You can use the indexing subroutines from [Subsection 10.4.2](http://math.hws.edu/eck/cs124/javanotes7/c10/s4.html#generics.4.2), modified to write the data to TextIO instead of to System.out. (You will need to make these subroutines static.) The input and output files should be selected by the user when the program is run. The sample program [*WordCount.java*](http://math.hws.edu/eck/cs124/javanotes7/source/chapter10/WordCount.java), from [Subsection 10.4.4](http://math.hws.edu/eck/cs124/javanotes7/c10/s4.html#generics.4.4), can be used as a model of how to use files. That program also has a useful subroutine that reads one word from input.

As you read the file, you want to take each word that you encounter and add it to the concordance along with the current line number. Keeping track of the line numbers is one of the trickiest parts of the problem. In an input file, the end of each line in the file is marked by the newline character, '\n'. Every time you encounter this character, you have to add one to the line number. WordCount.java ignores ends of lines. Because you need to find and count the end-of-line characters, your program cannot process the input file in exactly the same way as does WordCount.java. Also, you will need to detect the end of the file. The function TextIO.peek(), which is used to look ahead at the next character in the input, returns the value TextIO.EOF at end-of-file, after all the characters in the file have been read.

Because it is so common, don't include the word "the" in your concordance. Also, do not include words that have length less than 3.

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c10/ex5-ans.html)

**Exercise 10.6:**

The sample program [*SimpleInterpreter.java*](http://math.hws.edu/eck/cs124/javanotes7/source/chapter10/SimpleInterpreter.java) from [Subsection 10.4.1](http://math.hws.edu/eck/cs124/javanotes7/c10/s4.html#generics.4.1) can carry out commands of the form "let variable = expression" or "print expression". That program can handle expressions that contain variables, numbers, operators, and parentheses. Extend the program so that it can also handle the standard mathematical functions sin, cos, tan, abs, sqrt, and log. For example, the program should be able to evaluate an expression such as sin(3\*x-7)+log(sqrt(y)), assuming that the variables x and y have been given values. Note that the name of a function must be followed by an expression that is enclosed in parentheses.

In the original program, a symbol table holds a value for each variable that has been defined. In your program, you should add another type of symbol to the table to represent standard functions. You can use the following nested enumerated type and class for this purpose:

private enum Functions { SIN, COS, TAN, ABS, SQRT, LOG }

/\*\*

\* An object of this class represents one of the standard functions.

\*/

private static class StandardFunction {

/\*\*

\* Tells which function this is.

\*/

Functions functionCode;

/\*\*

\* Constructor creates an object to represent one of

\* the standard functions

\* @param code which function is represented.

\*/

StandardFunction(Functions code) {

functionCode = code;

}

/\*\*

\* Finds the value of this function for the specified

\* parameter value, x.

\*/

double evaluate(double x) {

switch(functionCode) {

case SIN:

return Math.sin(x);

case COS:

return Math.cos(x);

case TAN:

return Math.tan(x);

case ABS:

return Math.abs(x);

case SQRT:

return Math.sqrt(x);

default:

return Math.log(x);

}

}

} // end class StandardFunction

Add a symbol to the symbol table to represent each function. The key is the name of the function and the value is an object of type *StandardFunction* that represents the function. For example:

symbolTable.put("sin", new StandardFunction(Function.SIN));

In SimpleInterpreter.java, the symbol table is a map of type *HashMap<String,Double>*. It's not legal to use a *StandardFunction* as the value in such a map, so you will have to change the type of the map. The map has to hold two different types of objects. The easy way to make this possible is to create a map of type *HashMap<String,Object>*. (A better way is to create a general type to represent objects that can be values in the symbol table, and to define two subclasses of that class, one to represent variables and one to represent standard functions, but for this exercise, you should do it the easy way.)

In your parser, when you encounter a word, you have to be able to tell whether it's a variable or a standard function. Look up the word in the symbol table. If the associated object is non-null and is of type *Double*, then the word is a variable. If it is of type *StandardFunction*, then the word is a function. Remember that you can test the type of an object using the instanceof operator. For example: if (obj instanceof Double)

[See the Solution](http://math.hws.edu/eck/cs124/javanotes7/c10/ex6-ans.html)