# The Demo Programs in Folder Assignment5

**A. Arrays.** The concept of an array as a linear list of data, i.e., as the programming manifestation of a list, is not a difficult one. There is some new syntax associated with arrays, but that also should come easily. In learning about arrays, the harder concepts to master are

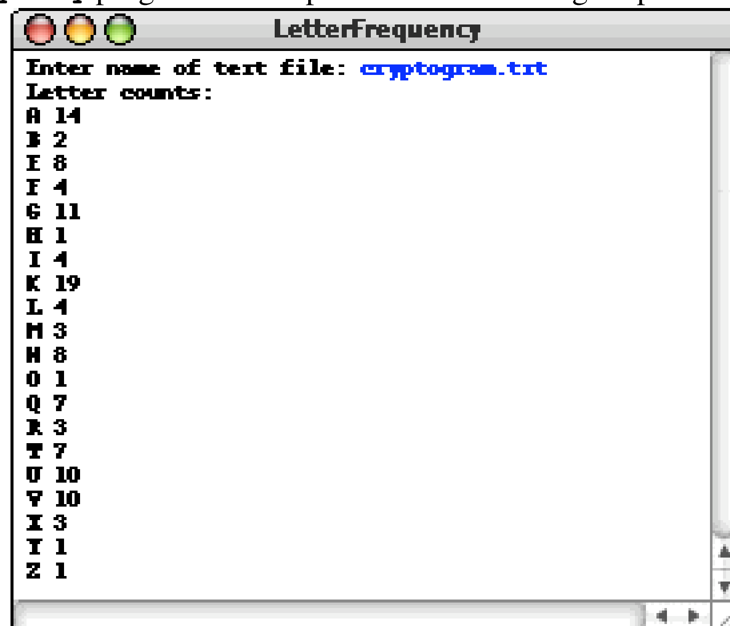
1. *How arrays are represented internally*. Understanding the internal representation of arrays is important for building your intuition about how array parameters work.
2. *How to use arrays effectively in applications*. Even though the mechanics of arrays are simple, it may not always be apparent when to use them. For applications that involve lists of data values, arrays come up naturally. There are, however, several circumstances in which the idea of using an array is harder to see.

The Roberts text includes an extensive discussion of the representation of arrays. This handout focuses on the second question in the context of a specific application that uses arrays in two rather different contexts.

The program **`LetterFrequency.java`** creates a table of the frequency of letters appearing in a text file. Such a program could be used in solving cryptograms, simple letter-substitution ciphers in which each letter in the plaintext message is replaced by some other letter. As an example, given a file containing the following cryptogram:

```
UTK UR VAKGK LKT NG HKTNBG VU VAK UVAKE;
QTF GU UR VAKGK. MANIA NG VAK TQVBEQO LQT,
QTF MANIA VAK GXNENV? MAU FKINXAKEG VAKL?
-- GAQZKGXKQEK, IULKFY UR KEEUEG
```

the **`LetterFrequency`** program should produce the following output for this file:



This program takes account of the following requirements:

- The program should count only the letters in the file. All other characters should be ignored.
- The program should ignore the case of each letter. Thus, if the file contains both an uppercase **A** and a lowercase **a**, the program should treat them as the same letter and make two entries in the corresponding counting array.
- The report at the end should be in order by letter and should not include any letters that do not appear in the cryptogram.

Letter-frequency programs are useful because they suggest what letters in the cryptogram might correspond to the most common letters in English. For example, in the sample run above, the characters **K**, **A**, and **G** each occur frequently and therefore might represent letters such as **E** or **T** in the original.

The program **List.java** illustrates by toy example how a FORTRAN programmer would implement list-management and garbage collection in the bad old days. Here the data are characters, so that part of memory is allocated as an array of characters. A second array, of integers, encodes the list of character array entries that is in use, and the list of character array entries that is free. Finally, individual integers point to the heads of the two lists. Since array indexing starts at 0, the value -1 connotes a null pointer.

**B. The ArrayList and HashMap Classes** Most of the power associated with modern, object-oriented languages such as Java does not come from the language itself but instead derives from the vast array of standard library classes that accompanies the language itself. For Java, that set of libraries is called the Java Development Kit, or JDK for short. The two classes ArrayList and HashMap form part of a much larger structure called the Java Collections Framework, which was added to the JDK as part of release 1.2 (we're now up to JDK version 1.5). As its name implies, the Java Collections Framework is designed to make it easier for programmers to work with collections of data of various kinds. The ArrayList class is designed to make it easier to work with ordered lists of data without having to take account of the low-level details of arrays. The HashMap class provides the ability to organize a collection, not by a linear order, but rather by setting up an association between a set of keys and a corresponding set of values.

Given any class in a toolkit like the JDK, it is possible to understand that class from a variety of perspectives. From a holistic perspective, each class is defined in terms by its abstract behavior. In this view, the important information about a class consists of the public methods it exports and a high-level model of what those methods do. From a reductionistic perspective, on the other hand, a class is defined in terms of its underlying implementation. At this level, the important information lies in the actual code for the methods it contains. If you are creating a class, you need to be concerned about each of these perspectives. In the more likely case that you are using a class as a tool, you can limit yourself to the more holistic, abstract view. In programming parlance, code that

makes use of a class—and by extension the programmers who write that code—are called **clients** of that class. As a general rule, clients can remain blissfully ignorant of the implementation of a class and concentrate solely on the **abstraction** it presents to the outside world.

In Math 121, our primary concern is to learn how to use classes as clients, but the algorithms and data structures course will look more deeply into how these classes are implemented and how clever algorithms make them as efficient as possible.

### The `ArrayList` class

A detailed description of this class appears in the Roberts text. The most important methods for the `ArrayList` class appear in Figure 10-4. This handout focuses on the relative advantages and disadvantages of `ArrayList`s with respect to Java arrays and, more specifically, the complexity of using wrapper classes to store primitive data values in an `ArrayList`.

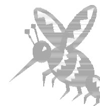
The most important advantages that `ArrayList`s offer over arrays are the following:

- An `ArrayList` can change its size dynamically without forcing the client to take care of the details. Once allocated, an array has a fixed length. Although it is possible to simulate the behavior of an `ArrayList` by reallocating a new array with the desired size and reassigning that value to the array variable, the client programmer must explicitly copy values from the old array into the new one.
- The fact that `ArrayList`s can grow and shrink dynamically makes it possible to define operations like `add` and `remove` that insert and delete elements. Java arrays offer no analogous operations.
- `ArrayList`s support a wide variety of additional operations beyond what is built into arrays. In particular, the `ArrayList` class includes the methods `indexOf` and `contains`, which allow clients to search an `ArrayList` for a particular value.

Although these advantages offer considerably more flexibility and power to the client, it is important to note that the `ArrayList` does have weaknesses, including the following:

- Using the `ArrayList` class is typically less efficient than working directly with the underlying arrays.
- The elements of an `ArrayList` are defined to be of class `Object`, which is the universal superclass for all other classes in Java. Because this class is invariably more general than the objects the client programmer stores in the `ArrayList`, it is almost always necessary to use a type cast when retrieving an object. As an example, suppose that you have created an `ArrayList` called `stringList` and used it to store values of type `String`. You could not retrieve the element at index position `i` simply by writing

```
String s = stringList.get(i);
```



because the value returned by get is an `Object` and is not yet known to be a `String`. You would instead need to code this statement using an explicit cast, as follows:

```
String s = (String) stringList.get(i);
```

- An **`ArrayList`** cannot hold a value of a primitive type such as **`int`** or **`char`** because such values are not objects. To get around this problem, you need to use the *wrapper classes* defined on page 179 of the text. Thus, to add a character **`ch`** to the end of an **`ArrayList`** called **`charList`**, you would need to write

  ```
  charList.add(new Character(ch));
  ```

  If you later wanted to retrieve the **`i`**th element from the list, you would need to use the rather cryptic expression

  ```
  ((Character) charList.get(i)).charValue()
  ```

The newest version of Java—Java 2 Standard Edition 5.0—contains features that largely eliminate the last two disadvantages, so things are getting better

### The `HashMap` class

Because it offers the same underlying data model as traditional arrays, the **`ArrayList`** class does not really offer much that seems new and exciting. In both structures, the elements of the collection are identified by a position number, which makes perfect sense if the data you are working with can be arranged in a linear order. In many applications, however, specifying a numeric index is not the ideal mechanism for identifying a particular element. It is often far more useful to associate data with an identifying value called a **key** that makes it easy to find that data in a large collection.

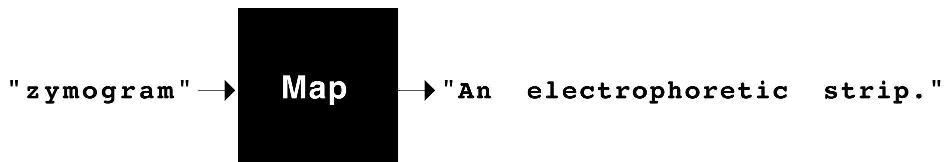
As an example, consider how you might represent a dictionary as a Java data structure. The fundamental property of a dictionary is that it associates words with definitions. Given a word, you'd like a simple way to retrieve its definition. You could, of course, store both the words and their corresponding definitions in arrays as shown in Figure 1. The **`words`** array would begin with the strings **`"a"`**, **`"aah"`**, and **`"aardvark"`** and end some 87,000+ entries later with **`"zymogram"`** and **`"zymosan"`**. The definitions could then be stored in a parallel array of strings called **`definitions`**. To look up a word in the dictionary, all you would need to do is find the word in the words **`array`** and then look up the corresponding index position in **`definitions`**.

**Figure 1. An array-based model of a dictionary**

| | words | definitions |
|---|---|---|
| 0 | a | The first letter of the English alphabet. |
| 1 | aah | To exclaim in amazement, joy, or surprise. |
| 2 | aardvark | A large burrowing nocturnal African mammal. |
| 3 | aardwolf | A maned striped mammal of southern Africa. |
| | | . . . |
| 87217 | zymogram | An electrophoretic strip. |
| 87218 | zymosan | An insoluble fraction of yeast cells. |

*Source: Webster's Ninth Collegiate Dictionary, 1983*

The strategy depicted in Figure 1, however, is entirely focused on the underlying representation and implementation. As a modern programmer, you are usually better off ignoring those low-level considerations and focusing instead on finding some class in the library that has the necessary abstract behavior. Here, all you need is a structure that can transform words into their corresponding definitions. How it does so might be interesting at some level—and indeed the `HashMap` class represents one of the cleverest ideas in an algorithms and data structures course—but entirely unnecessary for clients of the package. Thus, you need to find a structure that provides the following "black box" function:




In Java, the structure that you need is called a **map** . Conceptually, a map provides an association between **keys** (in this case, the words) and a corresponding **value** (in this case, the definitions). The fundamental operations on a map are

**put(***key***,** *value***)**

which adds a new association to the map linking *key* and *value,* and

**get(***key***)**

which returns the value associated with *key*.

The Java Collections Framework does export a construct with the generic name of `Map`. That structure, however, is not a class but an interface. As such, it merely defines the operations that a map must implement. When you want to make use of the functionality provided by maps, you have to choose one of the specific classes that implement the `Map` interface. The Java Collections Framework includes several of these,

**Figure 2. Common methods in the `HashMap` class**

| |
|---|
| **`HashMap()`**<br>Creates a new **`HashMap`** with no entries. |
| **`void put(Object key, Object value)`**<br>Adds an association between **`key`** and **`value`**, replacing any previous association for **`key`**. |
| **`Object get(Object key)`**<br>Returns the current association for **`key`**, or **`null`** if none exists. The result usually requires a cast. |
| **`void remove(Object key)`**<br>Removes the current association for **`key`**, if any. |
| **`boolean containsKey(Object key)`**<br>Returns **`true`** if the map contains an association for **`key`**. |
| **`int size()`**<br>Returns the number of key/value pairs in the **`HashMap`**. |
| **`Set keySet()`**<br>Returns a set of all the keys in the **`HashMap`**. |

but by far the most common is the **`HashMap`** class, which implements the map operations using a highly efficient and very clever strategy called a hash table. The most common operations on the **`HashMap`** class appear in Figure 2.

As with the **`ArrayList`** class, the **`HashMap`** class uses the universal **`Object`** class for the values that it stores. As a result, you need to use similar strategies when storing and retrieving values from a **`HashMap`**. In particular:

- When you retrieve a value from a **`HashMap`** using **`get`**, you generally need to use a type cast to convert it to a more specific type.
- If you want to use primitive types either as keys or as values, you need to use the wrapper classes introduced in Chapter 7. We'll see an example of this strategy in the class problem.

The only method from Figure 2 that requires additional explanation is the **`keySet`** method at the end of the list. This method returns the set of all keys in the **`HashMap`**, which makes it possible to determine what keys are defined. Unfortunately, it does so by using the **`Set`** interface, which is another part of the Java Collections Framework that lies beyond the scope of this course. For the time being, we will use that method only in the context of the following idiom, which iterates through all the keys in a **`HashMap`**:

```
for (Iterator i = map.keySet().iterator(); i.hasNext(); ) {
   type key = (type) i.next();
   . . . code to work with the key . . .
}
```

A simple example of this idiom is the following code, which displays all the key value pairs in a **`HashMap`** called **`dictionary`** in which the keys and values are both strings.

```
for (Iterator i = dictionary.keySet().iterator(); i.hasNext(); )
{
   String key = (String) i.next();
   String value = (String) dictionary.get(key);
   println(key + ": " + value);
}
```

**Trigraph frequencies**

> *Now, of all words in the language,* the *is most usual; let us see, therefore, whether there are not repetitions of any three characters, in the same order. . .*
>
> —Edgar Allan Poe, *The Gold Bug,*1843

We have seen a program that computed the frequency of letters, which is useful in decoding a letter-substitution cipher. As Poe's observation from *The Gold Bug* indicates, it is also useful to look at the **digraph** and **trigraph frequencies,** the number of times two or three letters appear together. In English, for example, the digraph `"th"` is very common, because it occurs in so many common words such as *the* and *this*. Poe's detective from *The Gold Bug* observes that `"the"` is itself the most common trigraph, and uses that fact to decode a message.

The program `TrigraphFrequency.java` reads a file of text and displays a table of the trigraph frequencies that occur within it. For example, given the same cryptogram as before,

```
UTK UR VAKGK LKT NG HKTNBG VU VAK UVAKE;
QTF GU UR VAKGK. MANIA NG VAK TQVBEQO LQT,
QTF MANIA VAK GXNENV? MAU FKINXAKEG VAKL?
-- GAQZKGXKQEK, IULKFY UR KEEUEG
```

the trigraph frequencies are given in the following table:

| | | | | |
|---|---|---|---|---|
| **AKE** → 2 | **FKI** → 1 | **KGK** → 2 | **NBG** → 1 | **UEG** → 1 |
| **AKG** → 2 | **GAQ** → 1 | **KGX** → 1 | **NEN** → 1 | **ULK** → 1 |
| **AKL** → 1 | **GXK** → 1 | **KIN** → 1 | **NIA** → 2 | **UTK** → 1 |
| **ANI** → 2 | **GXN** → 1 | **KQE** → 1 | **NXA** → 1 | **UVA** → 1 |
| **AQZ** → 1 | **HKT** → 1 | **KTN** → 1 | **QEK** → 1 | **VAK** → 7 |
| **BEQ** → 1 | **INX** → 1 | **LKF** → 1 | **QTF** → 2 | **VBE** → 1 |
| **EEU** → 1 | **IUL** → 1 | **LKT** → 1 | **QVB** → 1 | **XAK** → 1 |
| **ENV** → 1 | **KEE** → 1 | **LQT** → 1 | **QZK** → 1 | **XKQ** → 1 |
| **EQO** → 1 | **KEG** → 1 | **MAN** → 2 | **TNB** → 1 | **XNE** → 1 |
| **EUE** → 1 | **KFY** → 1 | **MAU** → 1 | **TQV** → 1 | **ZKG** → 1 |

The most common digraph in this example is `"VAK"`, which occurs seven times, and does indeed correspond to `"THE"`.

For another example, the console program `FlightPlanner.java` reads in a file containing flight destinations from various cities, and then allows the user to plan a round-trip flight route. Here is a sample run of the program:

```
FlightPlanner
Welcome to Flight Planner!
Here's a list of all the cities in our database:
  San Jose
  San Francisco
  Anchorage
  New York
  Honolulu
  Denver
Let's plan a round-trip route!
Enter the starting city: New York
From New York you can fly directly to:
  Anchorage
  San Jose
  San Francisco
  Honolulu
Where do you want to go from New York? Anchorage
From Anchorage you can fly directly to:
  New York
  San Jose
Where do you want to go from Anchorage? San Jose
From San Jose you can fly directly to:
  San Francisco
  Anchorage
Where do you want to go from San Jose? San Francisco
From San Francisco you can fly directly to:
  New York
  Honolulu
  Denver
Where do you want to go from San Francisco? New York
The route you've chosen is:
New York -> Anchorage -> San Jose -> San Francisco -> New York
```

The flights come from a data file named **`flights.txt`** with the following format:

- Each line consists of a pair of cities separated by an arrow indicated by the two-character combination **`->`**, as in

```
New York -> Anchorage
```

- The file may contain blank lines for readability

The entire data file used to produce this sample run is as follows:

```
San Jose -> San Francisco
San Jose -> Anchorage

New York -> Anchorage
New York -> San Jose
New York -> San Francisco
New York -> Honolulu

Anchorage -> New York
Anchorage -> San Jose

Honolulu -> New York
Honolulu -> San Francisco

Denver -> San Jose

San Francisco -> New York
San Francisco -> Honolulu
San Francisco -> Denver
```

The program runs by:

- Reading in the flight information from the file **`flights.txt`**.
- Displaying the complete list of cities.
- Allowing the user to select a city from which to start.
- In a loop, printing out all the destinations from the current city, and prompting the user to select the next city.
- Once the user has selected a round-trip route (i.e., once the user has returned to the starting city), exiting from the loop and print out the route that was chosen.