HST 952

Computing for Biomedical Scientists

Lecture 8

Outline

- Vectors
- Streams, Input, and Output in Java
- Programming examples

Vectors

- We can think of a vector as an array that can get larger or smaller when a program is running
- Data structure - a construct that allows us to organize/aggregate data
- An array is a *static data structure*
- A vector is a *dynamic data structure*

Arrays versus Vectors

Arrays

Bad:

- Size is fixed when declared
- Inefficient storage: can use a partially full array, but space has been allocated for the full size
- If one more value needs to be added past the maximum size the array needs to be redeclared

Good:

- More efficient (faster) execution
- Elements can be of any type

Vectors

Good :

- Size is not fixed
- Better storage efficiency: a partially full vector may be allocated just the space it needs
- If one more value needs to be added past the maximum size the vector size increases automatically

Bad:

- Less efficient (slower) execution
- Elements must be class types (primitive types not allowed)

Using Vectors

- Vectors are not automatically part of Java
  - they are in the `util` library
  - you must import `java.util.*`
- Create a vector with an initial size of 20 elements:

  ```
  Vector v = new Vector(20);
  ```

Vector Initial Capacity vs. Efficiency

- Choosing the initial size of a vector is an example of a tradeoff
  - making it too large wastes allocated memory space
  - making it too small slows execution
    - it takes time to resize vectors dynamically
- Solution?
  - optimize one at the expense of the other
  - or make good compromises
    - choose a size that is not too big and not too small

Vector Syntax

- The idea is the same as for arrays, but the syntax is different
- As with arrays, the index must be in the range 0 to size-of-the-vector

Array: `a` is a `String` array

```
a[i] = "Hi, Mom!");

String temp = a[i];
```

Vector: `v` is a vector

```
v.setElementAt("Hi,
  Mom!", i);
String temp =
   (String)v.elementAt(i);
```

Instead of the index in brackets and = for assignment, use vector method `setElementAt` with two arguments, the value and the index

Use vector method `elementAt(int index)` to retrieve the value of an element

Note: the cast to `String` is required because the base type of vector elements is `Object`

Vector Methods

- The vector class includes many useful methods:
  - constructors
  - array-like methods, e.g. `setElementAt` & `elementAt`
  - methods to add elements
  - methods to remove elements
  - search methods
  - methods to work with the vector's size and capacity, e.g. to find its size and check if it is empty
  - a `clone` method to copy a vector
  - see section 10.1 of Savitch text for more details

More Details About Vectors

- Vectors put values in successive indexes
  - `addElement` is used to put initial values in a vector
  - new values can be added only at the next higher index

- You can use `setElementAt` to change the value stored at a particular index
  - `setElementAt` can be used to assign the value of an indexed variable only if it has been previously assigned a value with `addElement`

Base Type of Vectors

- The base type of an array is specified when the array is declared
  - all elements of arrays must be of the same type
- The base type of a vector is `Object`
  - elements of a vector can be of any class type
  - in fact, *elements of a vector can be of different class types*
  - it is usually best to have all elements in a vector be the same class type
  - to store primitive types in a vector they must be converted to a corresponding wrapper class

More Details About Vectors

- The following code looks very reasonable but will produce an error saying that the class `Object` does not have a method named `length`:

```
Vector v = new Vector()
String greeting = "Hi, Mom!";
v.addElement(greeting);
System.out.println("Length is " +
    (v.elementAt(0)).length());
```

- `String`, of course, does have a `length` method, but Java sees the type of `v.elementAt(0)` as `Object`, not `String`
- Solution? Cast `v.elementAt(0)` to `String`:

```
System.out.println("Length is " +
(String)(v.elementAt(0)).length();
```

Vector Size Versus Vector Capacity

- Be sure to understand the difference between *capacity* and *size* of a vector:
  - *capacity* is the declared size of the vector
    - the current maximum number of elements
  - *size* is the actual number of elements being used
    - the number of elements that contain valid values, not garbage
    - remember that vectors add values only in successive indexes
- Loops that read vector elements should be limited by the value of `size`, not `capacity`, to avoid reading garbage values

Increasing Storage Efficiency of Vectors

- A vector automatically increases its size if elements beyond its current capacity are added
- But a vector does not automatically decrease its size if elements are deleted
- The method `trimToSize()` shrinks the capacity of a vector to its current size so there is no extra, wasted space
  - the allocated space is reduced to whatever is currently being used
- To use storage more efficiently, use `trimToSize()` when a vector will not need its extra capacity later

More Details About Vectors

- The method `clone` is used to make a copy of a vector but its return type is `Object`, not `Vector`
  - of course you want it to be `Vector`, not `Object`

- So, what do you do?
  - Cast it to `Vector`

```
Vector v = new Vector(10);
Vector otherV;
otherV = v;
Vector otherV2 = (Vector)v.clone();
```

This just makes `otherV` another *name* for the vector `v` (there is only one copy of the vector object and it now has two names referring to the same location/address in memory)

This creates a *second* copy of `v` with a different name, `otherV2` and a different address in memory

Protecting Private Variables

- Be careful not to return addresses of private vector variables, otherwise calling methods can access them directly
  - "Information Hiding" is compromised
- To protect against it, return a copy of the vector
  - use `clone` as described in the previous slide
- But that's not all:
  - if the elements of the vector are class (and not primitive) types, they may not have been written to pass a copy
  - they may pass their address
  - so additional work may be required to fix the accessor methods (have accessor methods return clones)

Programming example

Input/Output (I/O) Overview

- In this context it is input to and output from programs
- Input can be from keyboard or a file
- Output can be to display (screen) or a file
- Advantages of file I/O
  - permanent copy
  - output from one program can be input to another
  - input can be automated (rather than entered manually)

Streams

- *Stream*: an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
  - it acts as a buffer between the data source and destination
- *Input stream*: a stream that provides input to a program
- *Output stream*: a stream that accepts output from a program
  - `System.out` is an output stream
  - `System.in` is an input stream
- A stream connects a program to an I/O object
  - `System.out` connects a program to the screen
  - `System.in` connects a program to the keyboard

Binary Versus Text Files

- *All* data and programs are ultimately just zeros and ones
  - each digit can have one of two values, hence *binary*
  - *bit* is one binary digit, *byte* is a group of eight bits
- In *text files*: the bits represent printable characters
  - one byte per character for ASCII, the most common code
  - for example, Java source files are text files
  - so is any file created with a "text editor"
- In b*inary files*: the bits represent other types of encoded information, such as executable instructions or numeric data
  - these files are easily read by the computer but not humans
  - they are *not* intelligible to a human when printed

Binary Versus Text Files

- Text files are more readable by humans
- Binary files are more efficient
  - computers read and write binary files more easily than text
- Java binary files are portable
  - they can be used by Java on different machines
  - Reading and writing binary files is normally done by a program
  - text files are used only to communicate with humans

Java Text Files

- Source files
- Occasionally input files
- Occasionally output files

Java Binary Files

- Executable files (created by compiling source files)
- Usually input files
- Usually output files

Text File I/O

- Important classes for text file **output** (to the file)
  - **`PrintWriter, FileWriter, BufferedWriter`**
  - **`FileOutputStream`**
- Important classes for text file **input** (from the file):
  - **`BufferedReader`**
  - **`FileReader`**
- Note that **`FileOutputStream`** and **`FileReader`** are used only for their constructors, which can take file names as arguments.
  - **`PrintWriter`** and **`BufferedReader`** cannot take file names as arguments for their constructors.
- To use these classes your program needs a line like the following:

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

Every File Has Two Names

- The code to open the file creates two names for an output file
  - the name used by the operating system
    - e.g., `out.txt`
  - the stream name
    - e.g., `outputStream`
- Java programs use the ***stream*** name

Text File Output

- Binary files are more efficient for Java to process, but text files are readable by humans
- Java allows both binary and text file I/O
- To open a text file for output: connect a text file to a stream for writing
  - e.g., create a stream of the class `PrintWriter` and connect it to a text file

Text File Output

- For example:

```
PrintWriter outputStream = new PrintWriter(new
         FileOutputStream("out.txt"));
```

- Then you can use `print` and `println` to write to the file (convenient)
  - The text lists some other useful `PrintWriter` methods

Closing a File

- An output file should be closed when you are done writing to it (and an input file should be closed when you are done reading from it)
- Use the `close` method of the class
- If a program ends normally it will close any files that are open

Closing a file

If a program automatically closes files when it ends normally, why close them with explicit calls to `close`?

Two reasons:

1. To make sure it is closed if a program ends abnormally (the file could get damaged if it is left open).
2. A file that has been opened for writing must be closed before it can be opened for reading.

Text File Input

- To open a text file for input: connect a text file to a stream for reading
  - use a stream of the class `BufferedReader` and connect it to a text file
  - use the `FileReader` class to connect the `BufferedReader` object to the text file
- For example:

```
BufferedReader inputStream =
   new BufferedReader(new FileReader("data.txt"));
```

- Then:
  - read lines (`Strings`) with `BufferedReader`'s `readLine` method
  - `BufferedReader` has no methods to read numbers directly, so read numbers as `Strings` and then convert them
  - read a single `char` with `BufferedReader`'s `read method`

Input File Exceptions

- A `FileNotFoundException` is thrown if the file is not found when an attempt is made to open a file
- Most read methods throw `IOException`
  - we have to write a catch block for it
- If a read goes beyond the end of the file an `EOFException` is thrown

Handling IOException

- `IOException` cannot be ignored
  - either handle it with a catch block
  - or defer it with a `throws`-clause

Put code to open a file and read/write to it in a `try`-block and write a `catch`-block for this exception :

```
catch(IOException e)
{
   System.out.println("Problem…");
}
```

Testing for the End of an Input File

- A common programming situation is to read data from an input file but not know how much data the file contains
- In these situations you need to check for the end of the file
- There are three common ways to test for the end of a file:
  1. Put a sentinel value at the end of the file and test for it.
  2. Throw and catch an end-of-file exception.
  3. Test for a special character that signals the end of the file (text files often have such a character).

Testing for End of File in a Text File

- There are several ways to test for end of file. For reading text files in Java you can use this one:
  - Test for a special character that signals the end of the file
- When `readLine` tries to read beyond the end of a text file it returns the special value *`null`*
  - so you can test for `null` to stop processing a text file
- `read` returns -1 when it tries to read beyond the end of a text file
  - the `int` value of all ordinary characters is nonnegative

Programming example

Reading input from one file and writing output to another

Reading Parts of a String

- There are `BufferedReader` methods to read a line and a character, but not just a single word
- `StringTokenizer` can be used to parse a line into words
  - it is in the `util` library so you need to import `java.util.*`
  - some of its useful methods are shown in the text
    - e.g. test if there are more tokens
  - you can specify *delimiters* (the character or characters that separate words)
    - the default delimiters are "white space" (space, tab, and newline)

Example: StringTokenizer

- Display the words separated by any of the following characters: space, new line (\n), period (.) or comma (,).

```
String inputLine = KeyboardInput.readLine();
StringTokenizer wordFinder =
new StringTokenizer(inputLine, " \n.,");
//the second argument is a string of the 4 delimiters
while(wordFinder.hasMoreTokens())
{
   System.out.println(wordFinder.nextToken());
}
```

Entering "`Question,2b.or !tooBee.`" gives this output:

```
Question
2b
or
!tooBee
```

Warning: Overwriting a File

- Opening a file creates an empty file
- Opening a file creates a new file if one does not already exist
- Opening a file that already exists eliminates the old file and creates a new, empty one
  - data in the original file is lost
- How to test for the existence of a file and avoid overwriting it is covered in section 9.3 of the text, which discusses the `File` class

The File Class

- Acts like a wrapper class for file names
- A file name like `"out.txt"` has only `String` properties
- But a file name of type `File` has some very useful methods
  - `exists`: tests to see if a file already exists
  - `canRead`: tests to see if the operating system will let you read a file
- `FileInputStream` and `FileOutputStream` have constructors that take a `File` argument as well as constructors that take a `String` argument

Summary

- *Text files* contain strings of printable characters; they look intelligible to humans when opened in a text editor.
- *Binary files* contain numbers or data in non-printable codes; they look *un*intelligible to humans when opened in a text editor.
- Java can process both binary and text files for I/O

Summary

- Always check for the end of the file when reading from a file. The way you check for end-of-file depends on the method you use to read from the file.
- A file name can be read from the keyboard into a `String` variable and the variable used in place of a file name.

Programming example

Want to create a simple parser that can read a boolean expression typed from the keyboard of the form:

true and true
true and false
true or true
true or false
not true
not false, etc.

and print out the truth value of the expression

Read

- Chapter 9
- Chapter 10