CSE 11: Lecture 15

- ✔ Comment translation
- ✔ Sorting
- ✔ Searching
- ✔ Partially filled arrays
- ✔ Parallel arrays
- ✔ Ragged and triangular arrays
- ✔ Arrays and the Java type system
- ✔ The Vector class

(Reading: Savitch, Ch. 6 and Ch 10)

Using arrays

- ✔ As an example of array use...
  - ✗ Write a function that takes as argument an array of chars, and returns true if the array of chars is a palindrome; else false

    (a *palindrome* is a string that is the same forward or backward... so

```
palindrome("Hello there!".toCharArray())  // false

palindrome("ABLE WAS I ERE I SAW ELBA".toCharArray()) // true

palindrome("AAAABBBB".toCharArray())   // false

palindrome("Aabbaa".toCharArray())        // returns false

palindrome(StringToChars("1881"))          // returns true
```

Comment translation

- ✔ Let’s try the “comment translation” strategy for code development:
  - ✗ 1. In English pseudocode, write the algorithm the method will implement
  - ✗ 2. Put this English pseudocode in the body of the method with //’s before each line
  - ✗ 3. Translate the English pseudocode into Java
- ✔ The result: commented Java code that (hopefully) does what you want!

A palindrome-testing method: first step, pseudocode

```
Requirement: return true if the char array holds a
palindrome;else return false

Let n be the length of the char array

 For each i from 0 up to n...
Compare the ith element from the front of the
array to the ith element from the end.
If they are different, return false immediately


if the loop completes without return, we have a palindrome
```

A palindrome-testing method: second step, comments

```
// return true if the char array holds a palindrome; else false
boolean palindrome(char s[])
{
   // Let n be the length of the char array

   //   For each i from 0 up to n...
   // Compare the ith element from the front of the
   //  array to the ith element from the end.
   // If they are different, return false immediately


   // if the loop completes without return, we have a palindrome

}
```

✔ Now, this will compile... but it doesn’t do anything.

A palindrome-testing method: third step, translation

```
// return true if the char array holds a palindrome; else false
boolean palindrome(char s[])
{
   // Let n be the length of the char array
   int n = s.length;

   //   For each i from 0 up to n...
   for(int i=0; i<n; i++) {

      // Compare the ith element from the front of the
      //  array to the ith element from the end.
      // If they are different, return false immediately
      if (s[i] != s[n-i-1]) return false;
   }

   // if the loop completes without return, we have a palindrome
   return true;
}
```

✔ Result: commented code that works. However, it can be improved...

A slightly smarter palindrome-testing method...

- ✔ The previous method does about twice as many checks as necessary! This is better:

```
boolean palindrome(char[] s)
{
   int len = s.length;  // len is the length of the array
   int n = len / 2;  // n is "half" the length of the array

   // for each i from 0 to n...
   for(int i=0; i<n; i++) {

      // compare the ith element from the front of the
      // array to the ith element from the end.
      // if they are ever different, return false

      if (s[i] != s[len-i-1]) return false;
   }
   // if strlen(s) is odd, we don’t check the "middle" char...
   // but that’s okay, the middle char can be anything.
   return true;
}
```

Sorting

- ✔ Sorting is the process of re-arranging the elements of a sequence so that they are in order, according to some ordering relation
  - ✗ There are many algorithms for sorting; entire books have been written on this topic
  - ✗ Sorting is useful in many contexts: it can make subsequent human or machine processing of data much more efficient

- ✔ We will look ways to definite a sorting function with header

```
// PRE:  A is an array of doubles
// POST: the elements of A are in increasing order:
// A[0] <= A[1]  && A[1] <= A[2] &&...&& A[len-2] <= A[A.length-1]
void sort(double A[])
```

- ✔ We will first consider a simple (and not very efficient) algorithm called "bubble sort". It has the advantage of being relatively easy to understand. In pseudocode:

```
for (int N = 0; N < A.length; N++)
{
   "Bubble-up" the Nth smallest element of A to A[N]
   ("Bubbling-up" means exchanging adjacent elements of A)
}
```

Bubble sort

```
void sort(double A[])
// PRE:  A is an array of len doubles
// POST: the elements of A are in increasing order:
// A[0] <= A[1]  && A[1] <= A[2] && ... && A[len-2] <= A[len-1]
{
   int len = A.length;
   for (int N=0; N<len; N++) {

      // bubble up the smallest element in A[N],...,A[len-1]
      // by adjacent exchanges until it is in A[N]
      for (int M=len-1; M>N; M--) {
         if (A[M] < A[M-1]) {
            swap(A,M,M-1);
         }
      }
      // loop invariant:  At this point, the elements
      // A[0],...,A[N] are sorted

   }
}
```

Selection sort

- Bubble sort is a simple, easy to code and understand (but not very efficient) sorting algorithm
- We could make it somewhat more efficient in various ways...
- For example, instead of bubbling up the smallest element with index greater or equal to N by adjacent swaps until it is in A[N], we could search the elements with index greater than or equal to N, select the smallest, and swap it with A[N]
  - ... this can reduce the number of swaps, but number of comparisons stays the same
- Same interface, but the implementation will be different:

```
// PRE:  A is an array of doubles
// POST: the elements of A are in increasing order:
// A[0] <= A[1]  && A[1] <= A[2] &&...&& A[len-2] <= A[A.length-1]
void sort(double A[])
```

- Selection sort in pseudocode:

```
for (int N = 0; N < A.length; N++)
{
   Select the Nth smallest element of A;
   Swap it with A[N];
}
```

Selection sort

```
// PRE:  A is an array of len doubles
// POST: the elements of A are in increasing order:
// A[0] <= A[1]  && A[1] <= A[2] && ... && A[len-2] <= A[len-1]
void sort(double A[])
{
   int len = A.length;
   for (int N=0; N<len; N++) {
      // find the smallest element in A[N],...,A[len-1]
      // this is the Nth smallest element in the whole array
      int smallindx = N;
      double small = A[N];
      for (int M=N+1; M<len; M++) {
         if (A[M] < small) {
            smallindx = M;
            small = A[M];
         }
      }
      swap(A,N,smallindx); //... and swap it with A[N]
      // loop invariant:  the elements A[0],...,A[N] are sorted
   }
```

Sorting efficiency

- ✔ Selection sort and bubble sort are both easy sorting algorithms to understand, and to implement correctly... but:
- ✔ When sorting a typical sequence of length N, bubble sort and selection sort take a number of steps proportional to N^2
  - ✗ bubble sort and selection sort are therefore called “N-squared” or “quadratic” sorting algorithms
  - ✗ This may be okay if N is small (the sequence you are sorting is short), but for long sequences you should use a more efficient algorithm
- ✔ More sophisticated sorting algorithms will sort a typical sequence of length N using a number of steps proportional to N times the base-2 logarithm of N: for example, quicksort, heapsort, mergesort do this
  - ✗ quicksort, heapsort, and mergesort are therefore called “N log N” sorting algorithms
  - ✗ N log N is much better than N^2 for large N, and bubble sort or selection sort should not be used for long sequences

Searching

- Searching is the process of finding out whether a data structure holds an element with a certain value
- For example, you might have a database of book titles, and you want to know if the title "Manufacturing Consent" is in the database
- This search (also known as lookup or find) operation is obviously pretty important in general
- If your database is implemented as an array without anything special (e.g. sorting or hashing) there is no better search algorithm than *sequential* search
- Sequential search (also sometimes called linear search) just means going through the array in sequence starting from the beginning, comparing each element to what you're looking for, stopping either when you find it, or reach the end without finding it:

```
boolean find(Object arr[], Object key) {

   for(int i=0; i<arr.length; i++)
      if (arr[i].equals(key)) return true;

   return false;

}
```

Searching efficiency

- When searching for an item in a database of size N, linear search takes a number steps proportional to N, on average
  - This may be okay if N is small, but for large databases you should use a more efficient approach
- More efficient searching can involve more sophisticated data structures than arrays, as well as more sophisticated algorithms. Some alternatives:
  - Binary search in a sorted array: on average, number of steps proportional to log N
  - Tree search in a binary search tree: on average, steps proportional to log N
  - Hashing in a hash table: on average, number of steps constant, independent of N
- These faster sorting and searching algorithms will be covered in CSE 12, CSE 100, and CSE 101

Partially filled arrays

- When you create an array in Java, the elements of the array are initialized to default values: zero/false for primitive type elements, null for class type elements
- Until you store values in the elements of the array, the default initialized values are for most purposes not meaningful data
- An array that has only some of its elements storing meaningful values is called a *partially filled* array
- Every array in Java can tell you the number of elements it has (just look at its length instance variable) but if you are dealing with a partially filled array, you may need another variable to tell you how many elements are filled...

Partially filled arrays: an example

- ✓ The user will enter up to 1000 nonnegative integers at the terminal. A negative sentinel values indicates the end of the input. The program will print out how many numbers entered were larger than the average of the numbers entered.

```
int[] a = new int[1000];  // array to hold input values
int nFilled = 0;  // keeps track of how many numbers entered

do    a[nFilled++] = SavitchIn.readLineInt();
while (nFilled <= 1000 && a[nFilled-1] >= 0);
if (a[nFilled-1]<0) nFilled--;
// nFilled now is the number of integers read
// compute the average
int i=0, sum=0;
for(i=0; i<nFilled; i++) sum += a[i];
double avg = sum / (double) nFilled;

// print out how many numbers were larger than average
int nLarger = 0;
for(i=0; i<nFilled; i++) if(a[i] > avg) nLarger++;

System.out.println(nLarger + " were larger than average");
```

Parallel arrays

- ✔ An array is a sequence of variables, all of the same type
- ✔ Sometimes you want to have a sequence of elements of several different types
- ✔ For example, you might want to have a database of employees’ names (represented as Strings) and salaries (represented as doubles)
- ✔ It’s important to make sure that the different types of information for an employee (name and salary in this case) stay together
- ✔ One way to do this: Create an array of Employee objects

```
Employee record[] = new Employee[800];
```

where the Employee class is defined with instance variables like

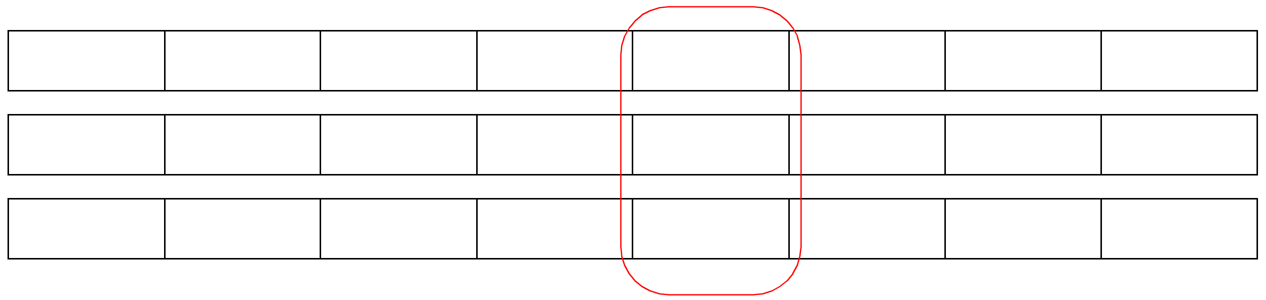
```
public class Employee {
   private String name;
   private double salary;
```

In this way `record[N]` should contain information about the Nth employee’s name and salary, kept together as instance variables of the object `record[N]` points to

- ✔ Another way to do it: use parallel arrays

Parallel arrays, cont'd

- Parallel arrays are a set of arrays containing related information
- The parallel arrays should have the same length
- The Nth element of the arrays contain data that belong together. This is why they are called parallel arrays:



- For example, in the employee database example, you might have

```
String name[] = new String[800];
double salary[] = new double[800];
```

- The idea: `name[N]` should contain information about the Nth employee's name, and `salary[N]` should contain information about the Nth employee's salary
- Parallel arrays can be useful, but in an OO language like Java they are not used that much

Rectangular, triangular, and ragged arrays

- ✔ The arrays-of-arrays we've seen so far are "rectangular" arrays: all the "rows" have the same length

- ✔ In Java, it is not necessary for arrays to be rectangular. A declaration like

```
int a[][];
```

says that the type of **a** is an array of arrays of ints; it doesn't say how many arrays of ints there are, or how many ints there are in any of the arrays

- ✔ It is easy to create a rectangular array, as we have seen:

```
a = new int[100][200];
```

This makes **a** point to an array of 100 arrays of ints each of which has the same length (200). Here's another way to create the same rectangular array:

```
a = new int[100][]; // a points to an array of 100 ptrs to int[]
for(int i=0; i<100; i++)  a[i] = new int[200];
```

- ✔ With that idea in mind, now let's look at creating non-rectangular arrays

Triangular arrays

- Triangular arrays often occur in numerical matrix computations
- An *upper-triangular* array is an array of N arrays. The first array has N elements; the second has N-1 elements; ... , and the last array has one element.
- A *lower-triangular* array is an array of N arrays. The first array has one element; the second has 2 elements; ... , and the last array has N elements.
- You can create an upper triangular array with 300 rows like this:

```
int t[][];  // t will be the triangular array
// first, create 300 pointers to arrays of ints, but
// don't create the arrays of ints themselves:
t = new int[300][];
// now create each of the arrays of ints, of appropriate length:
for(int r=0; r<300; r++)
   t[r] = new int[300-r];
```

- Lower triangular arrays can be created similarly...

Ragged arrays

- ✔ Ragged arrays are arrays of arrays which have a rather random distribution of lengths
- ✔ For example, suppose you wanted a 2-dimensional array to represent a calendar: the rows of the calendar would be months, and the columns would be days of the month. (If this is an array of arrays of doubles, each element might represent the amount of rainfall on that day.) You could create the array like this:

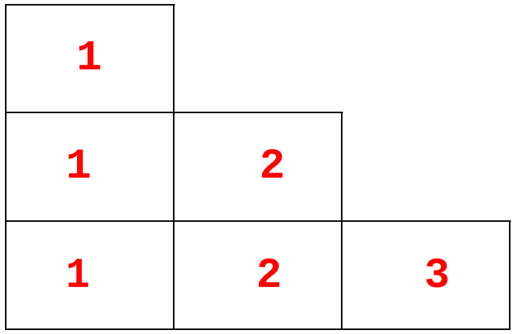
```
double calendar[][] = new double[12][];
calendar[0] = new double[31];  // January
calendar[1] = new double[28];  // February
calendar[2] = new double[31];  // March
calendar[3] = new double[30];  // April
calendar[4] = new double[31];  // May
calendar[5] = new double[30];  // June


// etc!
```

Nonrectangular arrays and initializers

- ✔ When initializing a multidimensional array with a curly-brace expression, the lengths of the arrays do not have to be the same. For example, you can create a lower-triangular array this way:

```
// create a as a lower-triangular array
int[][] a = {{1}, {1,2}, {1, 2, 3}};
```

the array a:



Arrays and the Java type system

- ✔ An array is a sequence of variables, all of the same type
- ✔ In Java, "of the same type" means: according to the Java type system, which pays attention to the is-a relationships established by class derivation
- ✔ In particular, if D is a class descended from a class B, then an element of an array-of-B can be a pointer to a D:

```
B[] arrB = new B[10];
arrB[0] = new D();  // okay:  every D is-a B
D[] arrD = new D[10];
arrD[0] = new B();  // compile-time error! not every B is-a D
```

- ✔ Also, arrays themselves follow the is-a relationship established by class derivation. If every D is-a B, then every array-of-D is-a array-of-B:

```
arrB = new D[33];   // okay:  every D[] is-a B[]
arrB[0] = new D();  // okay
arrB[1] = new B();  // run-time error!  storing a B in a D[]
```

- ✔ So, as a corollary, array of Object pointers can contain pointers to any Objects! This is the basis for "generic" data structures in Java

Arrays and Vectors

- In Java, an array has a length (number of elements) that is determined at the time the array is created

```
byte[] a;  // create a pointer to a byte array

a = new byte[22];  // create an array of 22 bytes
```

- Once created, you cannot change the size of a Java array, though you can create a new one of a different size, and copy the elements of the old one to the new one

```
byte[] b = new byte[44];
for(int i=0; i<a.length; i++) b[i] = a[i];
a = b;
```

- The java.util.Vector class provides a generic, variable-length sequential data structure that does this for you
- Vector uses a partially-filled `Object[]` array internally, creating a new one and copying when necessary

- Let's look at some methods of this class, to see how it works

A look at Vector.java

```
package java.util;

/**
 * The <code>Vector</code> class implements a growable array of
 * objects. Like an array, it contains components that can be
 * accessed using an integer index. However, the size of a
 * <code>Vector</code> can grow or shrink as needed to accommodate
 * adding and removing items after the <code>Vector</code> has been
 * created.
 */
public
class Vector implements Cloneable, java.io.Serializable {
```

Vector instance variables

```
/**
 * The array buffer into which the components of the vector are
 * stored. The capacity of the vector is the length of this array
 * buffer.
 */
 protected Object elementData[];

/**
 * The number of valid components in the vector.
 */
 protected int elementCount;

/**
 * The amount by which the capacity of the vector is automatically
 * incremented when its size becomes greater than its capacity. If
 * the capacity increment is <code>0</code>, the capacity of the
 * vector is doubled each time it needs to grow.
 */
 protected int capacityIncrement;
```

Vector constructors, and size()

```
 public Vector(int initialCapacity, int capacityIncrement) {
   super();
   this.elementData = new Object[initialCapacity];
   this.capacityIncrement = capacityIncrement;
 }
 public Vector() {
   this(10,0);
 }


/**
 * Returns the number of components in this vector.
 *
 * @return  the number of components in this vector.
 */
public final int size() {
   return elementCount;
}
```

add()

```
/**
 * Adds the specified component to the end of this vector,
 * increasing its size by one. The capacity of this vector is
 * increased if its size becomes greater than its capacity.
 *
 * @param   obj   the component to be added.
 * @since   JDK1.2
 */
public final synchronized void add(Object obj) {
  ensureCapacity(elementCount + 1);
  elementData[elementCount++] = obj;
}
```

- ✔ There is also a method `add(int index, Object obj)` that permits inserting a new element anywhere in the Vector...
- ✔ and a method `remove(int index)` that permits removing an element from anywhere in the Vector...
- ✔ and several other useful methods. But we will concentrate on how the underlying elementData array is changed, if its capacity is exceeded

ensureCapacity()

```
/**
 * Increases the capacity of this vector, if necessary, to ensure
 * that it can hold at least the number of components specified
 * by the minimum capacity argument.
 * @param   minCapacity   the desired minimum capacity.
 */
public final synchronized void ensureCapacity(int minCapacity) {
  int oldCapacity = elementData.length;
   if (minCapacity > oldCapacity) {
     Object oldData[] = elementData;
     int newCapacity = (capacityIncrement > 0) ?
       (oldCapacity + capacityIncrement) : (oldCapacity * 2);
     if (newCapacity < minCapacity) {
       newCapacity = minCapacity;
     }
     elementData = new Object[newCapacity];
     System.arraycopy(oldData, 0, elementData, 0, elementCount);
  }
}
```

✔ System.arraycopy() does a fast copy of array elements, using native code

Next time...

- ✔ Generic sorting and the java.util.Comparator interface
- ✔ Applets and HTML
- ✔ Javadoc

(Reading: Savitch, Ch. 13 for Applets, Appendix 10 for Javadoc)