Lecture 17 - 18

Multidimensional Arrays and the ArrayList Class

Dr. Martin O'Connor
CA166
www.computing.dcu.ie/~moconnor

Topics

- Declaring and Instantiating Multidimensional Arrays
- Aggregate Two-Dimensional Array Operations
- Higher Multidimensional Arrays
- The *ArrayList* Class

Two-Dimensional Arrays

- Allow organization of data in rows and columns in a table-like representation.

Example:

Daily temperatures can be arranged as 52 weeks with 7 days each.

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Week 1	35	28.6	29.3	38	43.1	45.6	49
Week 2	51.9	37.9	34.1	37.1	39	40.5	43.2
…							
…							
…							
…							
…							
Week 51	56.2	51.9	45.3	48.7	42.9	35.5	38.2
Week 52	33.2	27.1	24.9	29.8	37.7	39.9	38.8

Declaring Multidimensional Arrays

Declaring a two-dimensional array:

```
datatype [][] arrayName;
```

or

```
datatype [][] arrayName1, arrayName2, …;
```

Declaring a three-dimensional array:

```
datatype [][][] arrayName;
```

or

```
datatype [][][] arrayName1, arrayName2, …;
```

Examples:

```
double [][] dailyTemps, weeklyTemps;
Auto [][][] cars;
```

Instantiating Multidimensional Arrays

Instantiating a two-dimensional array:

```
arrayName = new datatype [exp1][exp2];
```

where exp1 and exp2 are expressions that evaluate to integers and specify, respectively, the number of rows and the number of columns in the array.

Example:

```
dailyTemps = new double [52][7];
```

dailyTemps has 52 rows and 7 columns, for a total of 364 elements.

Default Initial Values

When an array is instantiated, the array elements are given standard default values, identical to the default values of single-dimensional arrays:

Array data type	Default value
byte, *short*, *int*, *long*	0
float, *double*	0.0
char	The null character
boolean	*false*
Any object reference (for example, a *String*)	*null*

Assigning Initial Values

```
datatype [][] arrayName =
    { { value00, value01, … },
      { value10, value11, …},  … };
```

where *valueMN* is an expression that evaluates to the data type of the array and is the value to assign to the element at row *M* and column *N*.

- The number of sublists determines the number of rows in the array.
- The number of values in each sublist determines the number of columns in that row.

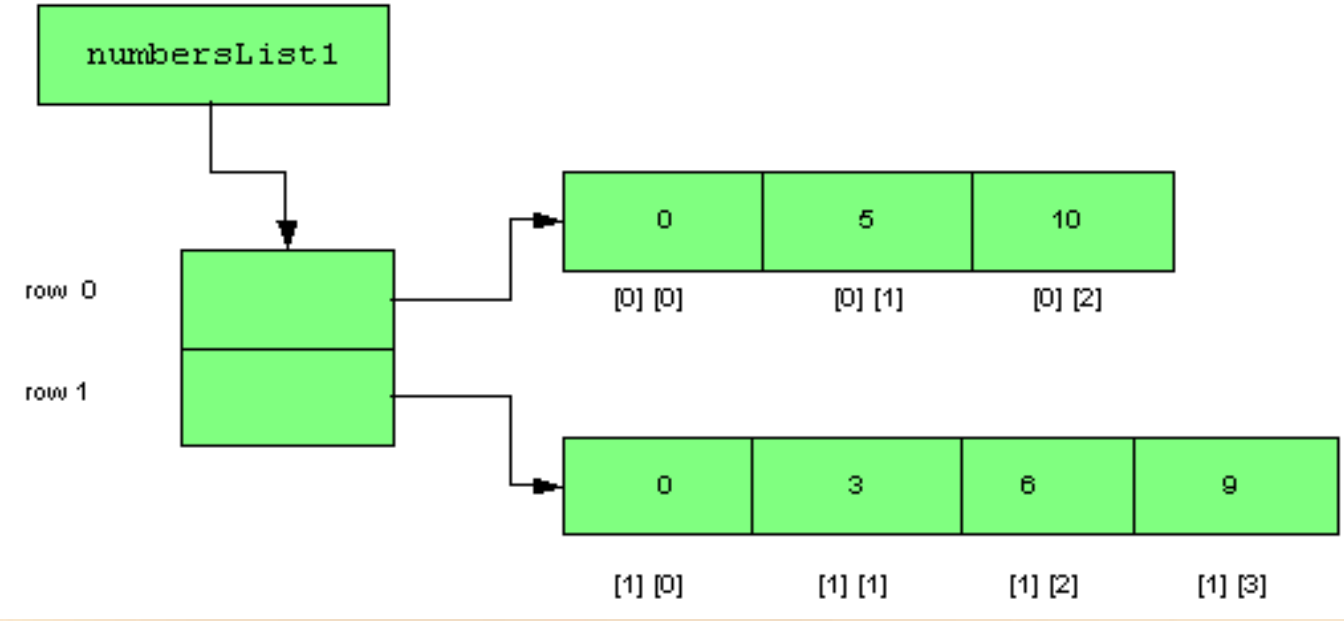
Thus, a two-dimensional array can have a different number of columns in each row.

Assigning Initial Values Example

For example, this statement:

```
int [][] numbersList1 = { { 0, 5, 10 },
                          { 0, 3, 6, 9 } };
```

instantiates this array:



An Array of Arrays

As the preceding figure illustrates, a two-dimensional array is an array of arrays.

- The first dimension of a two-dimensional array is an array of array references, with each reference pointing to a single-dimensional array.
- Thus, a two-dimensional array is comprised of an array of rows, where each row is a single-dimensional array.

Instantiating 2-D Arrays with Different-Length Rows

To instantiate a two-dimensional array whose rows have a different number of columns:

1. instantiate the two-dimensional array
2. instantiate each row as a single-dimensional array

```
//instantiate the array with 3 rows
char [][] grades = new char [3][];
// instantiate each row
grades[0] = new char [23];  // instantiate row 0
grades[1] = new char [16];  // instantiate row 1
grades[2] = new char [12];  // instantiate row 2
```

Accessing Array Elements

Elements of a two-dimensional array are accessed using this syntax:

```
arrayName[exp1][exp2]
```

- *exp1* is the element's row index.
  - row index of the first row: 0
  - row index of last row: number of rows - 1
- *exp2* is the element's column index.
  - column index of first column: 0
  - column index of last column: number of columns in that row - 1

The Length of the Array

The number of rows in a two-dimensional array is:

```
arrayName.length
```

The number of columns in row *n* in a two-dimensional array is:

```
arrayName[n].length
```

Accessing Two-Dimensional Array Elements

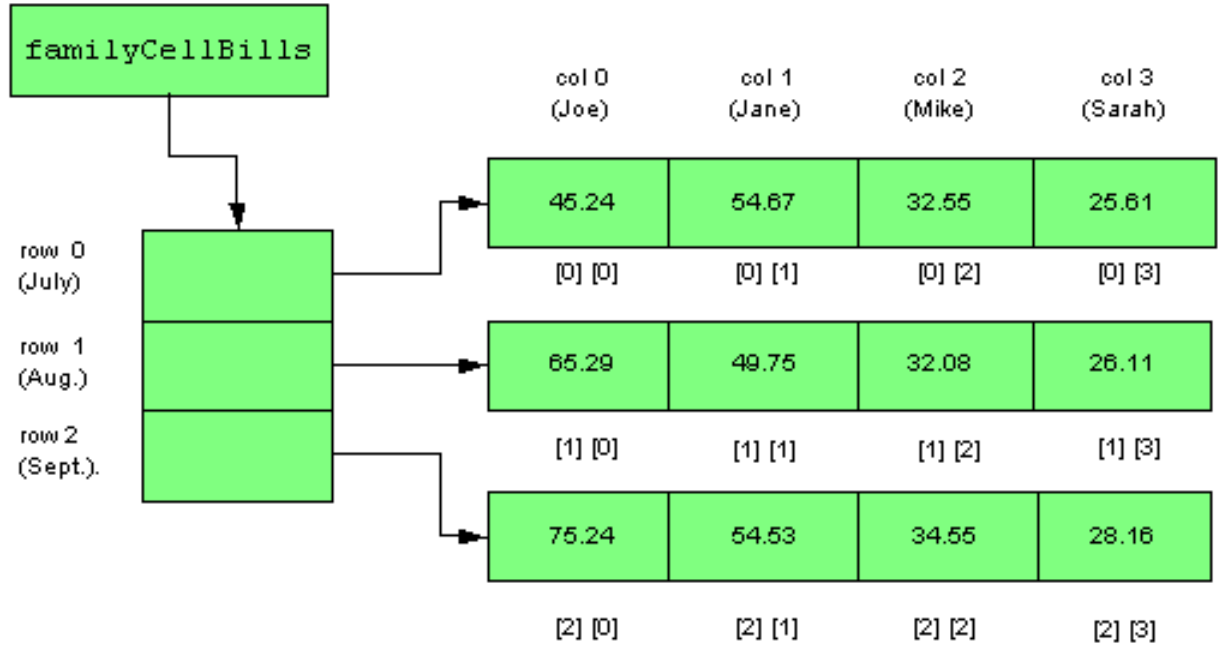
Array element	Syntax
Row 0, column *j*	`arrayName[0][j]`
Row *i*, column *j*	`arrayName[i][j]`
Last row, column *j*	`arrayName[arrayName.length - 1][j]`

Last row, last column	`arrayName[arrayName.length - 1]`
	`  [arrayName`
	`  [arrayName.length -1].length - 1]`

Number of rows	`arrayName.length`
Number of columns in row *i*	`arrayName[i].length`

Example: Family Cell Bills

- We want to analyze three months of cell phone bills for a family of four:



Aggregate Array Operations

To process all array elements in row order, we use a nested *for* loop:

```
for ( int i = 0; i < arrayName.length; i++ )
{
  for ( int j = 0; j < arrayName[i].length; j++ )
  {
     // process element arrayName[i][j]
  }
}
```

- The outer loop processes the rows.
- The inner loop processes the columns within each row.

Processing a Given Row

If we want to find the maximum bill for a particular month or the total bills for a month, we need to process just one row.

To process just row *i*, we use this standard form:

```
for ( int j = 0; j < arrayName[i].length; j++ )
{
   // process element arrayName[i][j]
}
```

Processing a Given Column

If we want to process the cell bills for one person only, we need to process just one column.

To process just column *j*, we use this standard form:

```
for ( int i = 0; i < arrayName.length; i++ )
{
    if ( j < arrayName[i].length )
        // process element arrayName[i][j]
}
```

Note: Because rows have variable lengths, we must verify that the current row has a column *j* before attempting to process the element.

Processing all Rows and Columns at a Time

If we want to determine the total of the cell bills for each month, we need to process all rows, calculating a total at the end of each row.

We use this standard form:

```
for ( int i = 0; i < arrayName.length; i++ )
{
  // initialize processing variables for row i
  for ( int j = 0; j < arrayName[i].length; j++ )
  {
      // process element arrayName[i][j]
  } // end inner for loop
  // finish the processing of row i
} // end outer for loop
```

Common Error Trap

Failing to initialize the row processing variables before processing each row is a logic error and will generate incorrect results.

Processing a Column at a Time

- Suppose we want to store test grades for three courses. Each course has a different number of tests, so each row has a different number of columns:

```
int [][] grades = { { 89, 75 },
                    { 84, 76, 92, 96 },
                    { 80, 88, 95 } };
```

- First, we need to find the number of columns in the largest row. We use that in our loop condition.
- Then before attempting to process the array elements, we check whether the given column exists in the current row.

Processing a Column at a Time (con't)

- Assume we have stored the maximum number of columns in *maxNumberOfColumns*; then the general pattern for processing elements one column at a time is:

```
for ( int j = 0; j < maxNumberOfColumns; j++ )
{
  for ( int i = 0; i < arrayName.length; i++ )
  {
    // does column j exist in this row?
    if ( j < arrayName[i].length )
    {
       // process element arrayName[i][j]
    }
  }
}
```

Two-Dimensional Arrays Passed to and Returned from Methods

- The syntax for a method that accepts a 2-D array as a parameter is the following:

```
returnType methodName(arrayType[ ][ ]
                      arrayParameterName)
```

- The syntax for a method that returns a 2-D array is the following:

```
returnArrayType[ ][ ] methodName(paramList)
```

- The caller of the method passes the argument list and assigns the return value to a reference to a 2-D array of the appropriate data type.

Other Multidimensional Arrays

If we want to keep track of sales on a per-year, per-week, and per-day basis, we could use a three-dimensional array:

- 1st dimension: year
- 2nd dimension: week
- 3rd dimension: day of the week

3-D Array
Sample Code

```
// declare a three-dimensional array
double [][][] sales;

// instantiate the array for 10 years, 52 weeks,
//  and 7 days
sales = new double [10][52][7];

// set the value of the first element
sales[0][0][0] = 638.50;

// set the value for year 5, week 23, day 4
sales [4][22][3] = 928.20;

// set the last value in the array
sales [9][51][6] = 1234.90;
```

Structure of an *n*-Dimensional Array

Dimension	Array Element
first	*arrayName*$[i_1]$ is an (n-1)-dimensional array
second	*arrayName*$[i_1][i_2]$ is an (n-2)-dimensional array
k^{th}	*arrayName*$[i_1][i_2][i_3][..][i_k]$ is an (n-k)-dimensional array
$(n-1)^{th}$	*arrayName*$[i_1][i_2][i_3][..][i_{n-1}]$ is a single-dimensional array
n^{th}	*arrayName*$[i_1][i_2][i_3][..][i_{n-1}][i_n]$ is an array element

General Pattern for Processing a Three-Dimensional Array

```
for ( int i = 0; i < arrayName.length; i++ )
{
  for ( int j = 0; j < arrayName[i].length; j++ )
  {
    for ( int k = 0; k < arrayName[i][j].length; k++ )
    {
      // process the element arrayName[i][j][k]
    }
  }
}
```

Code to Print the *sales* Array

```
for ( int i = 0; i < sales.length; i++ )  {
  for ( int j = 0; j < sales[i].length; j++ ) {
    for ( int k = 0; k < sales[i][j].length; k++ ) {

      // print the element at sales[i][j][k]
      System.out.print( sales[i][j][k] + "\t" );
    }
    // skip a line after each week
    System.out.println("");
  }
  // skip a line after each year
  System.out.println( );
}
```

A Four-Dimensional Array

If we want to keep track of sales on **a per-country,** per-year, per-week, and per-day basis, we could use a four-dimensional array:

- 1st dimension: country (assume 150 countries)
- 2nd dimension: year (assume 10 years)
- 3rd dimension: week
- 4th dimension: day of the week

```
double[][][][] sales = new double [150][10][52][7];
```

General Pattern for Processing a Four-Dimensional Array

```
for ( int i = 0; i < arrayName.length; i++ )
{
 for ( int j = 0; j < arrayName[i].length; j++ )
 {
  for ( int k = 0; k < arrayName[i][j].length; k++ )
  {
   for ( int l = 0; l < arrayName[i][j][k].length; l++ )
   {
     // process element arrayName[i][j][k][l]
   }
  }
 }
}
```

The *ArrayList* Class

- Arrays have a fixed size after they have been instantiated.
- What if we don't know how many elements we will need? For example, if we are
    - reading values from a file
    - returning search results
- We could create a very large array, but then we waste space for all unused elements.
- A better idea is to use an *ArrayList*, which stores elements of object references and automatically expands its size, as needed.

The *ArrayList* Class

- The *ArrayList* class is in the package: *java.util* (So it must be imported!)
- All *ArrayList* elements are object references, so we could have an *ArrayList* of *Auto* objects, *Book* objects, *Strings*, etc.
- To store primitive types in an *ArrayList*, use the wrapper classes (*Integer*, *Double*, *Character*, *Boolean*, etc.)
- The *ArrayList* is a generic class.
  - The *ArrayList* class has been written so that it can store object references of any type specified by the client.

Declaring an ArrayList

Use this syntax:

```
ArrayList<E> arrayListName;
```

E is a class name that specifies the type of object references that will be stored in the *ArrayList*.

Example:

```
ArrayList<String> listOfStrings;
ArrayList<Auto> listOfCars;
ArrayList<Integer> listOfInts;
```

ArrayList Constructors

Constructor name and argument list
`ArrayList<E>` constructs an *ArrayList* object of type *E* with an initial capacity of 10
`ArrayList<E>( int initialCapacity )` constructs an *ArrayList* object of type *E* with the specified initial capacity

- The capacity of an *ArrayList* is the total number of elements allocated to the list.
- The size of an an *ArrayList* is the number of elements that are used.

Instantiating an *ArrayList*

This list has a capacity of 10 *Astronaut* references, but a size of 0.

```
ArrayList<Astronaut> listOfAstronauts =
      new ArrayList<Astronaut>( );
```

This list has a capacity of 5 *Strings*, but a size of 0.

```
ArrayList<String> listOfStrings =
      new ArrayList<String>( 5 );
```

ArrayList Methods

Return value	Method name and argument list
`boolean`	`add( E element )` appends *element* to the end of the list
`void`	`add( int index, E element )` Inserts the specified element at the specified position in this list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices)
`void`	`clear( )` removes all the elements in the list
`int`	`size( )` returns the number of elements in the list
`E`	`remove( int index )` removes and returns the element at the specified *index* position. Shifts all elements after that position (if any) to the left (removes one from their indices)

More *ArrayList* Methods

Return value	Method name and argument list
E	`get( int index )` returns the element at the specified *index* position; the element is not removed from the list.
E	`set( int index, E element )` replaces the current *element* at the specified *index* position with the specified element and returns the replaced element.
int	`indexOf(Object o )` Returns index of first occurrence of specified element in this list (or -1 if not present)
Void	`trimToSize( )` sets the capacity of the list to its current size.

Sample Code – Array Lists

```
import java.util.ArrayList;

public class ArrayListDemo {
   public static void main(String args[]) {
      // create an array list
      ArrayList<String> demoArrList = new ArrayList<String>();
      System.out.println("Initial size of demoArrList: "
            + demoArrList.size());

      // add elements to the array list
      demoArrList.add("A");
      demoArrList.add("B");
      demoArrList.add("C");
      demoArrList.add("D");
      demoArrList.add(1, "E");
      System.out.println("Size of demoArrList after additions: "
            + demoArrList.size());

      // display the array list
      System.out.println("Contents of demoArrList: " + demoArrList);
      // Remove elements from the array list
      demoArrList.remove("C");
      demoArrList.remove(2);
      System.out.println("Size of demoArrList after deletions: "
            + demoArrList.size());
      System.out.println("Contents of demoArrList: " + demoArrList);
   }
}
```

Sample Code – Array Lists

Output is:

```
Initial size of demoArrList: 0
Size of demoArrList after additions: 5
Contents of demoArrList: [A, E, B, C, D]
Size of demoArrList after deletions: 3
Contents of demoArrList: [A, E, D]
```

- Lookup the Java API (java.util.ArrayList) for the full list of methods and constructors available for this class.

Processing Array Lists

Using a standard *for* loop:

```
ClassName objectIdentifier;
for ( int i = 0; i < arrayListName.size( ); i++ )
{
   currentObject = arrayListName.get( i );
   // process currentObject
}
```

Example:

```
Auto currentAuto;
for ( int i = 0; i < listOfAutos.size( ); i++ )
{
    currentAuto = listOfAutos.get( i );
    // process currentAuto
}
```

The Enhanced *for* Loop

Simplifies processing of lists.

The standard form is:

```
for ( ClassName currentObject : arrayListName )
{
   // process currentObject
}
```

This enhanced *for* loop prints all elements of an *ArrayList* of *Strings* named *secretList*:

```
for ( String s : secretList)
{
   System.out.println( s );
}
```

ArrayList Passed to and Returned from Methods

- The syntax for a method that accepts an ArrayList as a parameter is the following:

```
returnType methodName(ArrayList<ClassName>
                                arrayListParameterName)
```

- The syntax for a method that returns an ArrayList is the following:

```
ArrayList<ClassName> methodName(paramList)
```

ArrayList Passed to and Returned from Methods

- An example of passing an ArrayList as a parameter to a method

```
public static void addFlavors(ArrayList<String> flavorList) {
    flavorList.add("Vanilla");
    flavorList.add("Blueberry");
}
```

- An example of returning an ArrayList from a method

```
public ArrayList<Integer> getSecretCodes() {
    return this.secretCodes;
    // where secretCodes may have been declared in the class as
    // ArrayList<Integer> secretCodes = new ArrayList<Integer>();
}
```