Chapter 9

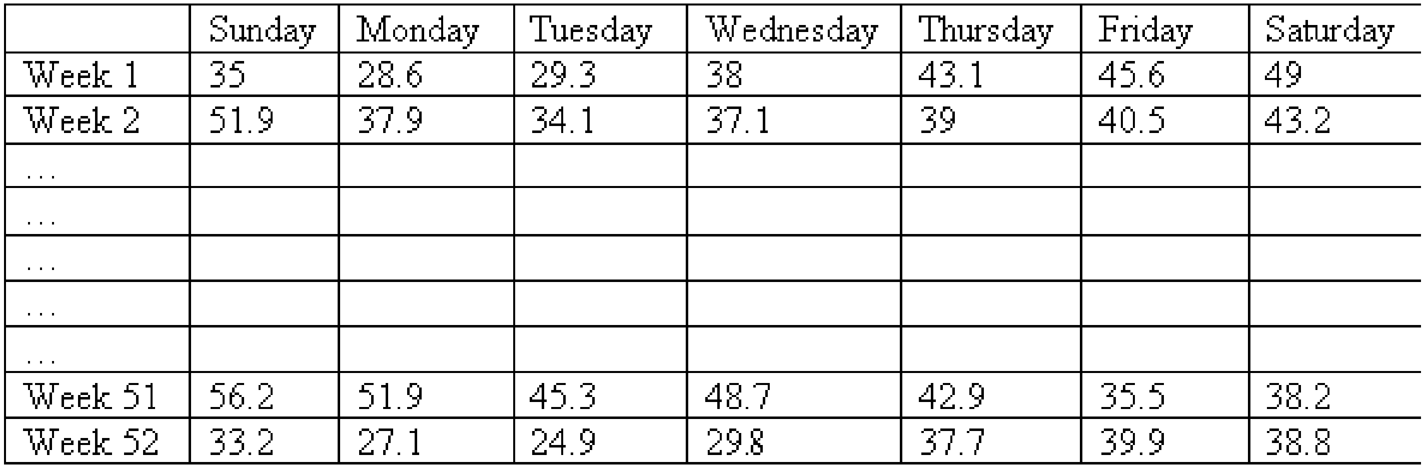
Multidimensional Arrays
and the
ArrayList Class

Topics

- Declaring and Instantiating Multidimensional Arrays
- Aggregate Two-Dimensional Array Operations
- Other 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 default values of single-dimensional arrays:

Array data type	**Default value**
byte, short, int, long	0
float, double	0.0
char	space
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.
```

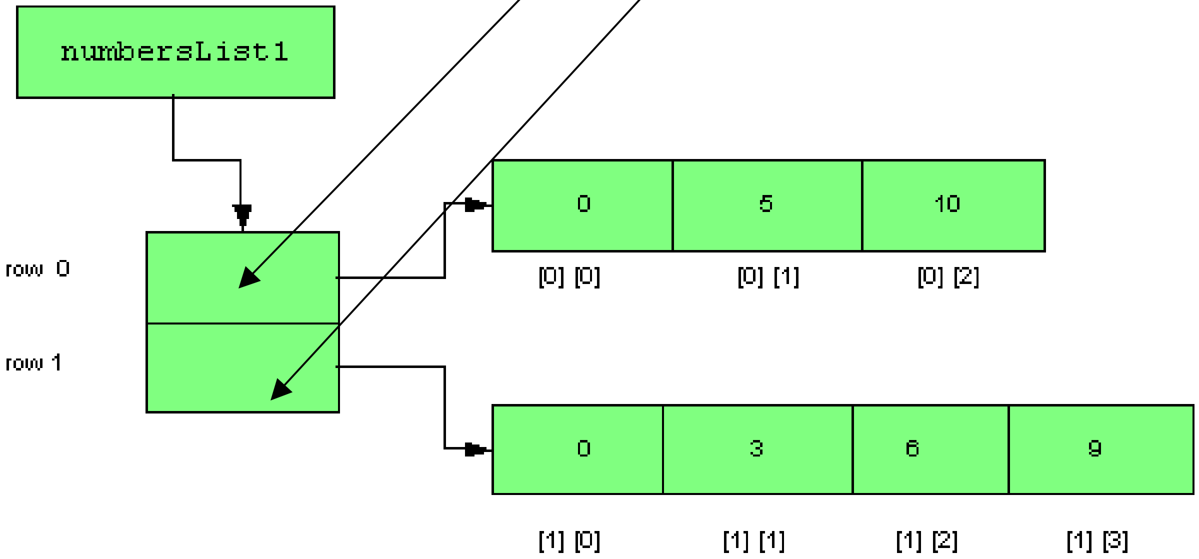
- The number of sublists is the number of rows in the array.
- The number of values in each sublist determines the number of columns in that row.
- Thus, a two-dimensional array can have a different number of columns in each row.

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 Arrays with Rows of Different Length

- To instantiate a two-dimensional array with a different number of columns for each row:

  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 position, or **row index.**
  - row index of first row: 0
  - row index of last row: number of rows - 1
- *exp2* is the element's column position, or **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:

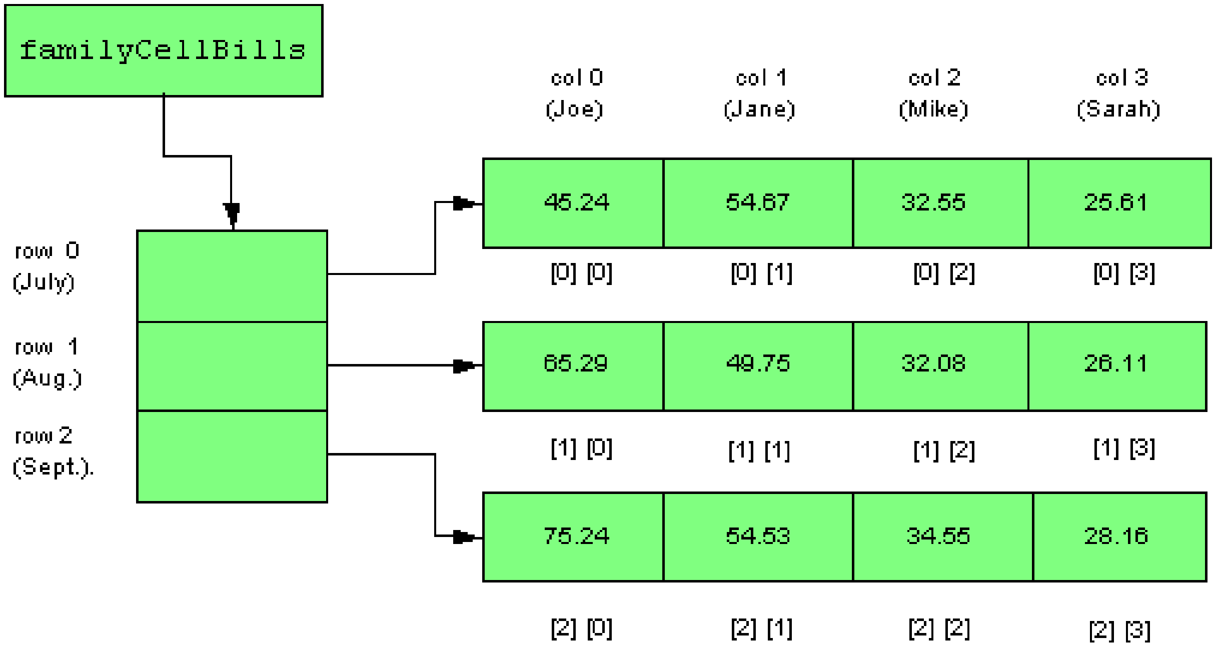
  $$\underbrace{\texttt{arrayName[n]}}_{\text{array}}\texttt{.length}$$

Summary: 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:



	col 0 (Joe)	col 1 (Jane)	col 2 (Mike)	col 3 (Sarah)
row 0 (July)	45.24 [0] [0]	54.67 [0] [1]	32.55 [0] [2]	25.61 [0] [3]
row 1 (Aug.)	65.29 [1] [0]	49.75 [1] [1]	32.08 [1] [2]	26.11 [1] [3]
row 2 (Sept.).	75.24 [2] [0]	54.53 [2] [1]	34.55 [2] [2]	28.16 [2] [3]

- *See Example 9.1 FamilyCellBills.java*

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.
- *See Example 9.3 OutputFamilyCellBills.java*

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]
}
```

- *See Example 9.4 SumRowFamilyCellBills.java*

Processing a Given Column

- If we want to determine the highest cell bill for one person, 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]
}
```

- Because rows have variable lengths, we must verify that the current row has a column *j* before attempting to process the element.
- *See Example 9.5 MaxMemberBill.java*

Processing One Row 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
```

- *See Example 9.6 SumEachRowFamilyCellBills.java*

The *ArrayList* Class

- Arrays have a fixed size once 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

- Package: *java.util*

- 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.)

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*

- For example:

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

- 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.

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 those 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 has 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`	`clear( )` removes all the elements in the list
`int`	`size( )` returns the number of elements
`E`	`remove( int index )` removes the element at the specified *index* position

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 *element* at the specified *index* position with the specified element
`void`	`trimToSize( )` sets the capacity of the list to its current size

Processing Array Lists

- Using a standard *for* loop:

```
ClassName currentObject;
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 *list*:

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

- *See Example 9.12 ArrayListOfIntegers.java*

Using an *ArrayList*

- We want to write a program for a bookstore that allows users to search for books using keywords.
- We will have three classes in this program:
  - A *Book* class, with instance variables representing the title, author, and price
  - A *BookStore* class that stores *Book* objects in an *ArrayList* and provides a *searchForTitle* method
  - A *BookSearchEngine* class, which provides the user interface and the *main* method
- *See Examples 9.13, 9.14, & 9.15*

Backup Slides

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 element, we check whether the column exists in the current row.

Processing A Column at a Time(con't)

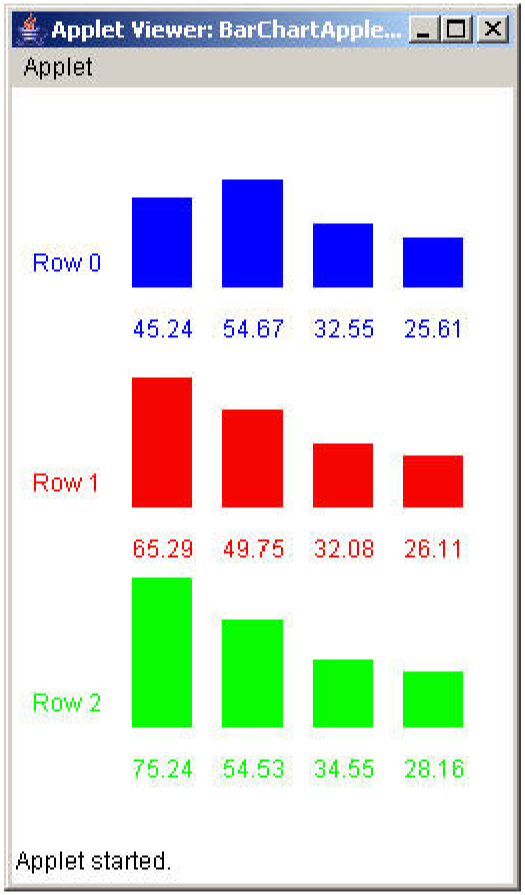
- We have stored the maximum number of columns in *maxNumberOfColumns*; the general pattern for processing elements one column at a time is:

```
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]
    }
  }
}
```

See Example 9.7 GradesProcessing.java

Displaying Array Data as a Bar Chart

- We use our standard nested *for* loops and the *fillRect* method of the *Graphics* class for the bars and the *drawString* method to print each element's value.
- To change colors for each row, we use an array of *Color* objects, and loop through the array to set the color for each row.



Applet started.

- Each time we process a row, we must reset the x and y values for the first bar.
- *See Example 9.8 BarChartApplet.java*

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

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 4, week 22, day 3
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
kth	$arrayName[i_1][i_2][i_3][..][i_k]$ is an (n-k)-dimensional array
(n-1)th	$arrayName[i_1][i_2][i_3][..][i_{n-1}]$ is a single-dimensional array
nth	$arrayName[i_1][i_2][i_3][..][i_{n-1}][i_n]$ 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 *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 month
  System.out.println( );
}
```

A Four-Dimensional Array

- If we want to keep track of sales on **a per-state,** per-year, per-week, and per-day basis, we could use a four-dimensional array:
  - 1st dimension: state
  - 2nd dimension: year
  - 3rd dimension: week
  - 4th dimension: day of the week

```
double[][][][] sales = new double [50][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]
   }
  }
 }
}
```