Week 5

Arrays and Array Lists

(Chapter 7)

Chapter Goals

© traveler1116/iStockphoto.

- To collect elements using arrays and array lists
- To use the enhanced for loop for traversing arrays and array lists
- To learn common algorithms for processing arrays and array lists
- To work with two-dimensional arrays
- To understand the concept of regression testing

Arrays

- An array collects a sequence of values of the same type.
- Create an array that can hold ten values of type double:

```
new double[10]
```

  - The number of elements is the `length` of the array
  - The new operator constructs the array

- The type of an array variable is the type of the element to be stored, followed by `[]`.
- To declare an array variable of type `double[]`

```
double[] values; ❶
```

To initialize the array variable with the array:

```
double[] values = new double[10];❷
```

- By default, each number in the array is 0
- You can specify the initial values when you create the array

```
double[] moreValues = { 32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65 };
```

Arrays

- To access a value in an array, specify which “slot” you want to use
  - use the `[]` operator

```
values[4] = 35;
```

- The “slot number” is called an index.
- Each slot contains an element.
- Individual elements are accessed by an integer index `i`, using the notation `array[i]`.
- An array element can be used like any variable.

```
System.out.println(values[4]);
```

Arrays

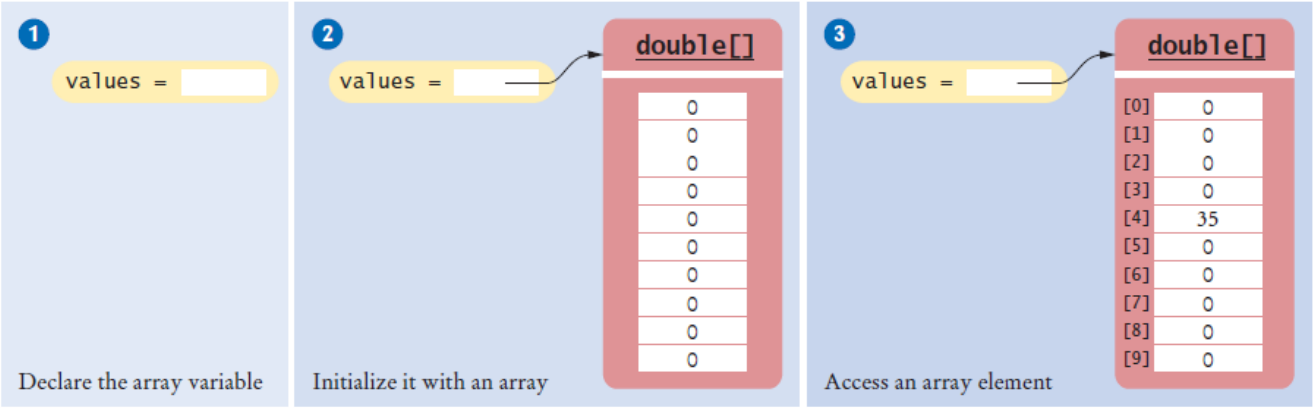


Figure 1 An Array of Size 10

Syntax 7.1 Arrays

Syntax

To construct an array:	`new` *typeName*[*length*]
To access an element:	*arrayReference*[*index*]

Name of array variable

Element type

Length

Type of array variable

```
double[] values = new double[10];

double[] moreValues = { 32, 54, 67.5, 29, 35 };
```

List of initial values

Use brackets to access an element.

```
values[i] = 0;
```

The index must be ≥ 0 and < the length of the array.

See page 314.

Arrays

- The elements of arrays are numbered starting at 0.
- The following declaration creates an array of 10 elements:

```
double[] values = new double[10];
```

- An index can be any integer ranging from 0 to 9.
- The first element is `values[0]`
- The last element is `values[9]`
- An array index must be *at least* zero and *less than* the size of the array.
- Like a mailbox that is identified by a box number, an array element is identified by an index.

© Luckie8/iStockphoto.

Arrays - Bounds Error

- A bounds error occurs if you supply an invalid array index.
- Causes your program to terminate with a run-time error.
- Example:

```
double[] values = new double[10];
values[10] = value; // Error
```

- `values.length` yields the length of the `values` array.
- There are no parentheses following `length`.

Declaring Arrays

Table 1 Declaring Arrays	
`int[] numbers = new int[10];`	An array of ten integers. All elements are initialized with zero.
`final int LENGTH = 10;` `int[] numbers = new int[LENGTH];`	It is a good idea to use a named constant instead of a "magic number".
`int length = in.nextInt();` `double[] data = new double[length];`	The length need not be a constant.
`int[] squares = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`String[] friends = { "Emily", "Bob", "Cindy" };`	An array of three strings.
`double[] data = new int[10];`	**Error:** You cannot initialize a `double[]` variable with an array of type `int[]`.

Array References

- An array reference specifies the location of an array.
- Copying the reference yields a second reference to the same array.
- When you copy an array variable into another, both variables refer to the same array

```
int[] scores = { 10, 9, 7, 4, 5 };
int[] values = scores; // Copying array reference
```

- You can modify the array through either of the variables:

```
scores[3] = 10;
System.out.println(values[3]); // Prints 10
```

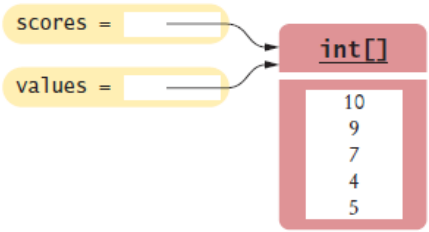


Figure 2 Two Array Variables Referencing the Same Array

Using Arrays with Methods

- Arrays can occur as method arguments and return values.
- An array as a method argument

```
public void addScores(int[] values)
{
  for (int i = 0; i < values.length; i++)
  {
    totalScore = totalScore + values[i];
  }
}
```

- To call this method

```
int[] scores = { 10, 9, 7, 10 };
fred.addScores(scores);
```

- A method with an array return value

```
public int[] getScores()
```

Partially Filled Arrays

- Array length = maximum number of elements in array.
- Usually, array is partially filled
- Define an array larger than you will need

```
final int LENGTH = 100;
double[] values = new double[LENGTH];
```

- Use companion variable to keep track of current size: call it `currentSize`

Partially Filled Arrays

- A loop to fill the array

```
int currentSize = 0;
Scanner in = new Scanner(System.in);
while (in.hasNextDouble())
{
  if (currentSize < values.length)
  {
    values[currentSize] = in.nextDouble();
    currentSize++;
  }
}
```

- At the end of the loop, `currentSize` contains the actual number of elements in the array.
- Note: Stop accepting inputs when `currentSize` reaches the array length.

Partially Filled Arrays

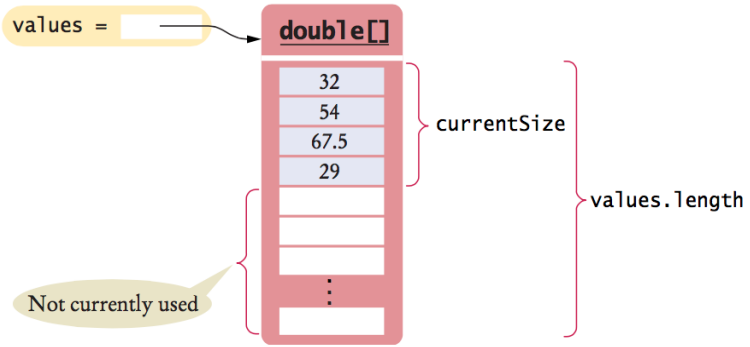


Figure 3 A partially-filled array

Partially Filled Arrays

- To process the gathered array elements, use the companion variable, not the array length:

```
for (int i = 0; i < currentSize; i++)
{
   System.out.println(values[i]);
}
```

- With a partially filled array, you need to remember how many elements

Make Parallel Arrays into Arrays of Objects

- Don't do this

```
int[] accountNumbers;
double[] balances;
```

- Don't use parallel arrays

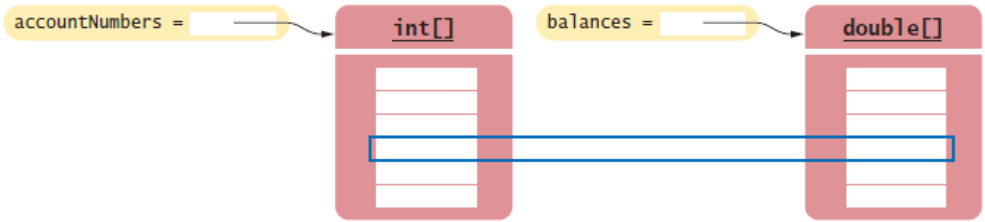


Figure 4 Avoid Parallel Arrays

Make Parallel Arrays into Arrays of Objects

- Avoid parallel arrays by changing them into arrays of objects:

```
BankAccount[] accounts;
```

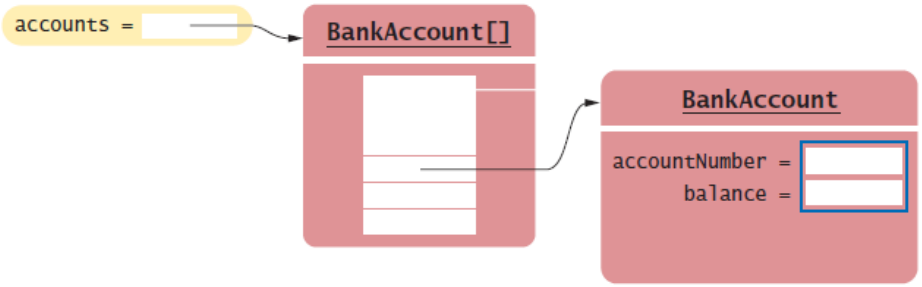


Figure 5 Reorganizing Parallel Arrays into an Array of Objects

The Enhanced `for` Loop

- You can use the enhanced `for` loop to visit all elements of an array.
- Totaling the elements in an array with the enhanced `for` loop

```
double[] values = . . .;
double total = 0;
for (double element : values)
{
    total = total + element;
}
```

- The loop body is executed for each element in the array `values`.
- Read the loop as “for each `element` in `values`.”
- Traditional alternative:

```
for (int i = 0; i < values.length; i++)
{
  double element = values[i];
  total = total + element;
}
```

The Enhanced `for` Loop

- Not suitable for all array algorithms.
- Does not allow you to modify the contents of an array.
- The following loop does not fill an array with zeros:

```
for (double element : values)
{
  element = 0; // ERROR: this assignment does not modify array elements
}
```

- Use a basic `for` loop instead:

```
for (int i = 0; i < values.length; i++)
{ values[i] = 0; /* OK */ }
```

- Use the enhanced `for` loop if you do not need the index values in the loop body. The enhanced `for` loop is a convenient mechanism for traversing all elements in a collection.

Syntax 7.2 The Enhanced "for" Loop

Syntax

```
for (typeName variable : collection)
{
   statements
}
```

This variable is set in each loop iteration.
It is only defined inside the loop.

An array

```
for (double element : values)
{
   sum = sum + element;
}
```

These statements are executed for each element.

The variable contains an element, not an index.

Common Array Algorithm: Filling

- Fill an array with squares (0, 1, 4, 9, 16, ...):

```
for (int i = 0; i < values.length; i++)
{
  values[i] = i * i;
}
```

Common Array Algorithm: Sum and Average

- To compute the sum of all elements in an array:

```
double total = 0;
for (double element : values)
{
   total = total + element;
}
```

- To obtain the average:

```
double average = 0;
if (values.length > 0) { average = total / values.length; }
```

Common Array Algorithm: Maximum or Minimum

- Finding the maximum in an array

```
double largest = values[0];
for (int i = 1; i < values.length; i++)
{
  if (values[i] > largest)
  {
   largest = values[i];
  }
}
```

The loop starts at 1 because we initialize largest with `values[0]`.

- Finding the minimum: reverse the comparison.
- These algorithms require that the array contain at least one element.

Common Array Algorithm: Element Separators

- When you display the elements of an array, you usually want to separate them:

```
Ann | Bob | Cindy
```

- Note that there is one fewer separator than there are elements
- Print the separator before each element *except the initial one* (with index 0):

```
for (int i = 0; i < names.size(); i++)
{
  if (i > 0)
  {
    System.out.print(" | ");
  }
  System.out.print(names.value[i]);
}
```

- To print five elements, you need four separators.

Common Array Algorithm: Linear Search

- To find the position of an element:

  Visit all elements until you have found a match or you have come to the end of the array

- Example: Find the first element that is equal to 100

```
int searchedValue = 100;
int pos = 0;
boolean found = false;
while (pos < values.length && !found)
{
  if (values[pos] == searchedValue)
  { found = true; }
  else
  { pos++; }
}
if (found)
{
 System.out.println("Found at position: " + pos);
}
else {
 System.out.println("Not found");
}
```

Common Array Algorithm: Linear Search

- This algorithm is called a **linear search**.
- A linear search inspects elements in sequence until a match is found.
- To search for a specific element, visit the elements and stop when you encounter the match.

Common Array Algorithm: Removing an Element

Problem: To remove the element with index `pos` from the array `values` with number of elements `currentSize`.

Unordered

1. Overwrite the element to be removed with the last element of the array.
2. Decrement the `currentSize` variable.

```
values[pos] = values[currentSize - 1];
currentSize--;
```

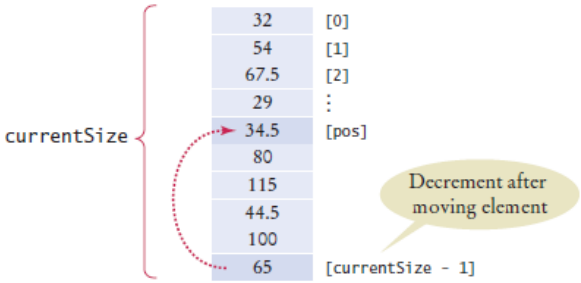


Figure 6 Removing an Element in an Unordered Array

Common Array Algorithm: Removing an Element

Ordered array

1. Move all elements following the element to be removed to a lower index.
2. Decrement the variable holding the size of the array.

```
for (int i = pos + 1; i < currentSize; i++)
{
  values[i - 1] = values[i];
}
currentSize--;
```

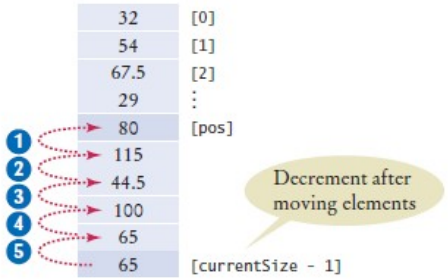


Figure 7 Removing an Element in an Ordered Array

Common Array Algorithm: Inserting an Element

- If order does not matter
  1. Insert the new element at the end of the array.
  2. Increment the variable tracking the size of the array.

```
if (currentSize < values.length)
{
  currentSize++
  values[currentSize - 1] = newElement;
}
```

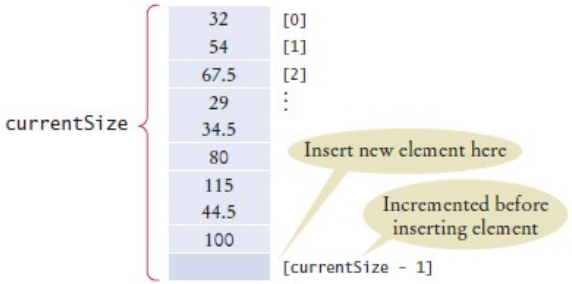


Figure 8 Inserting an Element in an Unordered Array

Common Array Algorithm: Inserting an Element

If order matters

1. Increment the variable tracking the size of the array.
2. Move all elements after the insertion location to a higher index.
3. Insert the element .

```
if (currentSize < values.length)
{
  currentSize++;
  for (int i = currentSize - 1; i > pos; i--)
  {
    values[i] = values[i - 1];
  }
  values[pos] = newElement;
}
```

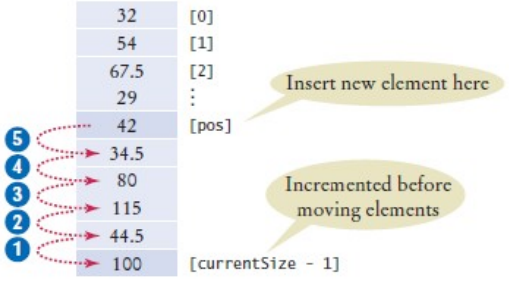
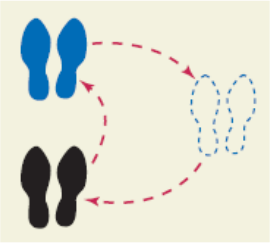


Figure 9 Inserting an Element in an Ordered Array

Common Array Algorithm: Swapping Elements

- To swap two elements, you need a temporary variable.



  - We need to save the first value in the temporary variable before replacing it.

```
double temp = values[i];
values[i] = values[j];
```

  - Now we can set `values[j]` to the saved value.

```
values[j] = temp;
```

Common Array Algorithm: Swapping Elements

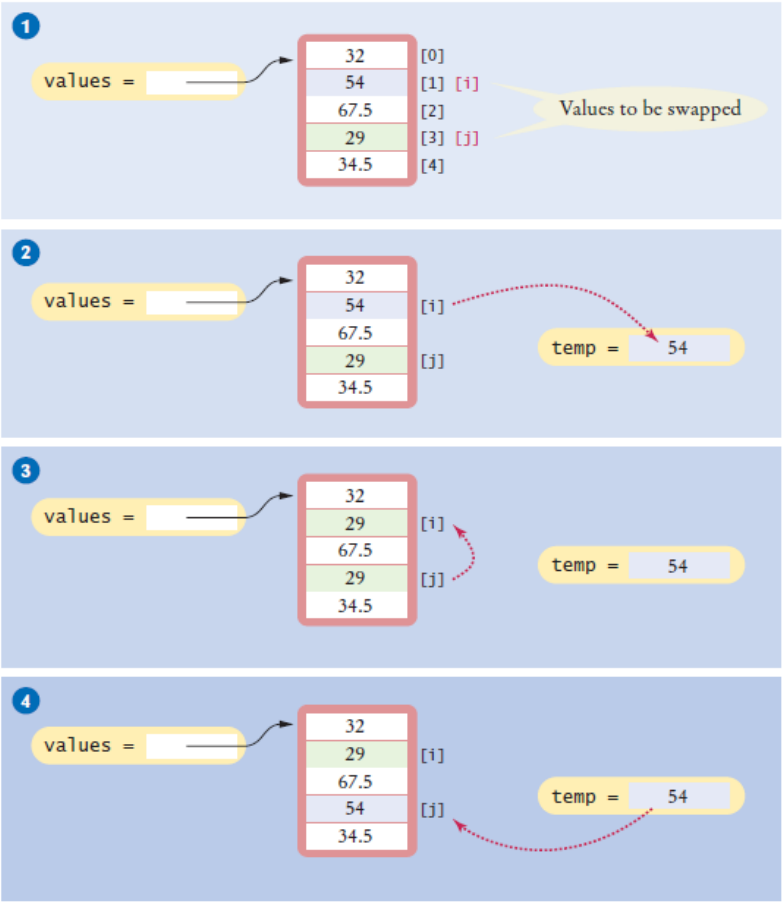


Figure 10 Swapping Array Elements

Common Array Algorithm: Copying an Array

- Copying an array variable yields a second reference to the same array:

```
double[] values = new double[6];
. . .  // Fill array
double[] prices = values; ❶
```

- To make a true copy of an array, call the `Arrays.copyOf` method:

```
double[] prices = Arrays.copyOf(values, values.length); ❷
```

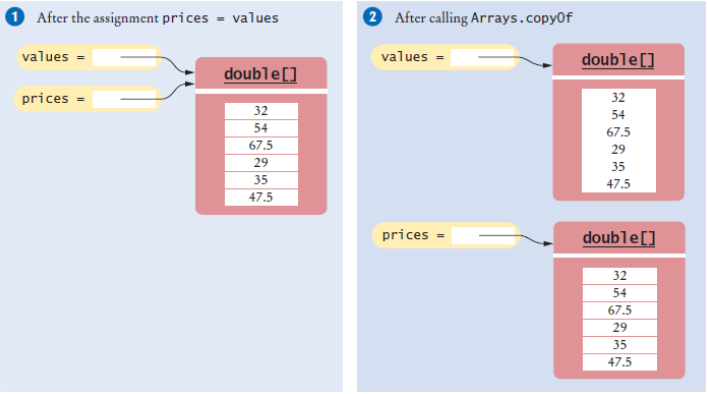


Figure 11 Copying an Array Reference versus Copying an Array

- To use the `Arrays` class, you need to add the following statement to the top of your program

```
import java.util.Arrays;
```

Common Array Algorithm: Growing an Array

- To grow an array that has run out of space, use the `Arrays.copyOf` method:
- To double the length of an array

```
double[] newValues = Arrays.copyOf(values, 2 * values.length); ❶
values = newValues; ❷
```

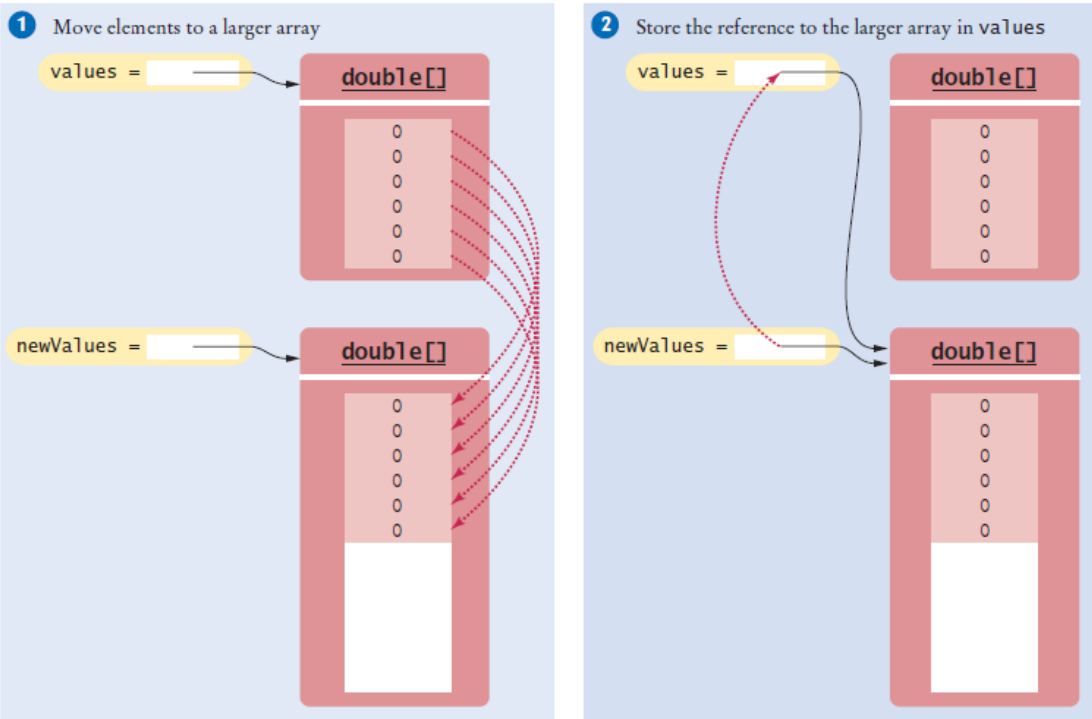


Figure 12 Growing an Array

Reading Input

- To read a sequence of arbitrary length:

  Add the inputs to an array until the end of the input has been reached.

  Grow when needed.

```
double[] inputs = new double[INITIAL_SIZE];
int currentSize = 0;
while (in.hasNextDouble())
{
  // Grow the array if it has been completely filled
  if (currentSize >= inputs.length)
  {
    inputs = Arrays.copyOf(inputs, 2 * inputs.length); // Grow the inputs array
  }
  inputs[currentSize] = in.nextDouble();
  currentSize++;
}
```

  Discard unfilled elements.

```
inputs = Arrays.copyOf(inputs, currentSize);
```

section_3/LargestInArray.java

```
1  import java.util.Scanner;
2
3  /**
4     This program reads a sequence of values and prints them, marking the largest value.
5  */
6  public class LargestInArray
7  {
8     public static void main(String[] args)
9     {
10       final int LENGTH = 100;
11       double[] values = new double[LENGTH];
12       int currentSize = 0;
13
14       // Read inputs
15
16       System.out.println("Please enter values, Q to
17       quit:");  Scanner in = new Scanner(System.in);
18       while (in.hasNextDouble() && currentSize <
19       values.length)
20       {
21          values[currentSize] = in.nextDouble();
22          currentSize++;
23       }
24       // Find the largest value
25
26       double largest = values[0];
27       for (int i = 1; i < currentSize; i++)
28       {
29          if (values[i] > largest)
30          {
31             largest = values[i];
32          }
```

Program Run:

```
Please enter values, Q to quit:
34.5 80 115 44.5 Q
34.5
80
115 <== largest value
44.5
```

Problem Solving: Adapting Algorithms

- By combining fundamental algorithms, you can solve complex programming tasks.
- Problem: Compute the final quiz score by dropping the lowest and finding the sum of all the remaining scores.
- Related algorithms:

  Calculating the sum Finding the minimum value Removing an element

- A plan of attack

```
Find the minimum.
Remove  it from the array.
Calculate the sum.
```

- Try it out. The minimum is 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	7	8.5	9.5	7	4	10

- We need to use a linear search to find the position of the minimum.

Problem Solving: Adapting Algorithms -continued

- Revise the plan of attack

```
Find the minimum value.
Find its position.
Remove that position from the array.
Calculate the sum.
```

- Try it out

Find the minimum value of 4. Linear search tells us that the value 4 occurs at position 5.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	7	8.5	9.5	7	4	10

Remove it

[0]	[1]	[2]	[3]	[4]	[5]
8	7	8.5	9.5	7	10

- Compute the sum: 8 + 7 + 8.5 + 9.5 + 7 + 10 = 50.
- This walk-through demonstrates that our strategy works.

Problem Solving: Adapting Algorithms -continued

- Inefficient to find the minimum and then make another pass through the array to obtain its position.
- Modify minimum algorithm to remember the **position** of the smallest element

```
int smallestPosition = 0;
for (int i = 1; i < values.length; i++)
{
  if (values[i] < values[smallestPosition])
  {
    smallestPosition = i;
  }
}
```

- Final Strategy

```
Find the position of the minimum.
Remove it from the array.
Calculate the sum.
```

Problem Solving: Discovering Algorithms by Manipulating Physical Objects

- Manipulating physical objects can give you ideas for discovering algorithms.

- The Problem: You are given an array whose size is an even number, and you are to switch the first and the second half.
- Example

This array

9	13	21	4	11	7	1	3

will become

11	7	1	3	9	13	21	4

Problem Solving: Discovering Algorithms by Manipulating Physical Objects

- Use a sequence of coins, playing cards, or toys to visualize an array of values.
- Original line of coins

- Removal of an array element

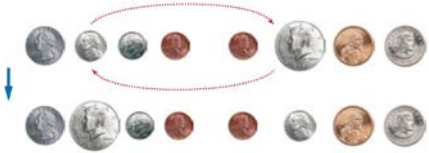
- Insertion of an array element

Problem Solving: Discovering Algorithms by Manipulating Physical Objects

- Swapping array elements

- Swap the coins in positions 0 and 4:

- Swap the coins in positions 1 and 5:



Problem Solving: Discovering Algorithms by Manipulating Physical Objects

- Two more swaps

- The pseudocode

```
i = 0
j = size / 2
While (i < size / 2)
  Swap elements at positions i and j
  i++
  j++
```

Two-Dimensional Arrays

- An arrangement consisting of rows and columns of values

  Also called a matrix.

- Example: medal counts of the figure skating competitions at the 2014 Winter Olympics.

	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

Figure 13 Figure Skating Medal Counts

- Use a two-dimensional array to store tabular data.
- When constructing a two-dimensional array, specify how many rows and columns are needed:

```
final int COUNTRIES = 8;
final int MEDALS = 3;
int[][] counts = new int[COUNTRIES][MEDALS];
```

Two-Dimensional Arrays

- You can declare and initialize the array by grouping each row:

```
int[][] counts =
{
   { 0, 3, 0 },
   { 0, 0, 1 },
   { 0, 0, 1 },
   { 1, 0, 0 },
   { 0, 0, 1 },
   { 3, 1, 1 },
   { 0, 1, 0 },
   { 1, 0, 1 }
};
```

- You cannot change the size of a two-dimensional array once it has been declared.

Syntax 7.3 Two-Dimensional Array Declaration

Number of rows

Name

Element type

Numberof columns

```
double[][] tableEntries = new double[7][3];
```

All values are initialized with 0.

Name

List of initial values

```
int[][] data = {
                { 16, 3, 2, 13 },
                { 5, 10, 11, 8 },
                { 9, 6, 7, 12 },
                { 4, 15, 14, 1 },
              };
```

Accessing Elements

- Access by using two index values, `array[i][j]`

```
int medalCount = counts[3][1];
```

- Use nested loops to access all elements in a two-dimensional array.
- Example: print all the elements of the `counts` array

```
for (int i = 0; i < COUNTRIES; i++)
{
  // Process the ith row
  for (int j = 0; j < MEDALS; j++)
  {
    // Process the jth column in the ith row
    System.out.printf("%8d", counts[i][j]);
  }
  System.out.println(); // Start a new line at the end of the row
}
```

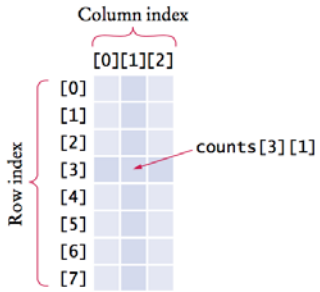


Figure 14 Accessing an Element in a Two-Dimensional Array

Accessing Elements

- Number of rows: `counts.length`
- Number of columns: `counts[0].length`
- Example: print all the elements of the `counts` array

```
for (int i = 0; i < counts.length; i++)
{
  for (int j = 0; j < counts[0].length; j++)
  {
    System.out.printf("%8d", counts[i][j]);
  }
  System.out.println();
}
```

Locating Neighboring Elements

[i - 1][j - 1]	[i - 1][j]	[i - 1][j + 1]
[i][j - 1]	[i][j]	[i][j + 1]
[i + 1][j - 1]	[i + 1][j]	[i + 1][j + 1]

Figure 15 Neighboring Locations in a Two-Dimensional Array

- Watch out for elements at the boundary array

  `counts[0][1]` does not have a neighbor to the top

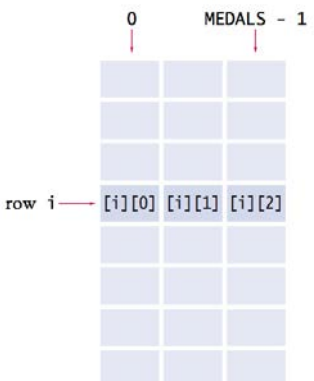
Accessing Rows and Columns

- Problem: To find the number of medals won by a country

  Find the sum of the elements in a row

- To find the sum of the ith row

  compute the sum of `counts[i][j]`, where j ranges from 0 to `MEDALS - 1`.



- Loop to compute the sum of the ith row

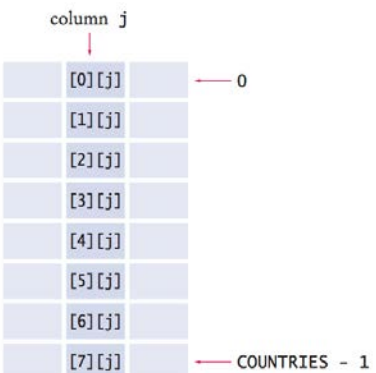
```
int total = 0;
for (int j = 0; j < MEDALS; j++)
{
  total = total + counts[i][j];
}
```

Accessing Rows and Columns

- To find the sum of the jth column

  Form the sum of `counts[i][j]`, where i ranges from 0 to `COUNTRIES - 1`

```
int total = 0;
for (int i = 0; i < COUNTRIES; i++)
{
 total = total + counts[i][j];
}
```



section_6/Medals.java

```
/**
    This program prints a table of medal winner counts with row  totals.
*/
public class Medals
{
   public static void main(String[] args)
   {
      final int COUNTRIES = 8;
      final int MEDALS = 3;

      String[] countries =
         {
            "Canada",
            "Italy",
            "Germany",
            "Japan",
            "Kazakhstan",
            "Russia",
            "South Korea",
            "United States"
         };

      int[][] counts =
         {
            { 0, 3, 0 },
            { 0, 0, 1 },
            { 0, 0, 1 },
            { 1, 0, 0 },
            { 0, 0, 1 },
            { 0, 1, 0 },
            { 0, 1, 0 },
            { 1, 0, 1 }
         };

      System.out.println("         Country      Gold  Silver   Bronze  Total");

      // Print countries, counts, and row totals
      for (int i = 0; i < COUNTRIES; i++)

```

Program Run:

```
       Country     Gold   Silver   Bronze    Total

        Canada        0        3        0        3
         Italy        0        0        1        1

       Germany        0        0        1        1

         Japan        1        0        0        1
    Kazakhstan        0        0        1        1
        Russia        3        1        1        5
   South Korea        0        1        0        1
United States        1        0        1        2
```

In Class Activity

- Work in groups of 2

- Write a class (which includes a swap method) that swaps the largest element with the smallest element of the given array. The required function prototype is given below:

  double[] swapArrayElement (double[] inComingArray)

- Write a test program (i.e., main function) to print the original as well as swapped arrays.

- Once completed exchange your paper with the group next to you and critique your answer.

Array Lists

- An array list stores a sequence of values whose size can change.
- An array list can grow and shrink as needed.
- `ArrayList` class supplies methods for many common tasks, such as inserting and removing elements.
- An array list expands to hold as many elements as needed.

Syntax 7.4 Array Lists

Syntax

To construct an array list:	`new ArrayList<`*typeName*`>()`
To access an element:	*arraylistReference*`.get(index)` *arraylistReference*`.set(index, value)`

Variable type

Variable name

An array list object of size 0

```
ArrayList<String> friends = new ArrayList<String>();
```

```
friends.add("Cindy");
String name = friends.get(i);
friends.set(i, "Harry");
```

The add method appends an element to the array list, increasing its size.

Use the get and set methods to access an element.

The index must be ≥ 0 and < friends.size().

Declaring and Using ArrayLists

- To declare an array list of strings

```
ArrayList<String> names = new ArrayList<String>();
```

- To use an array list

```
import java.util.ArrayList;
```

- `ArrayList` is a **generic class**
- Angle brackets denote a **type parameter**

  Replace `String` with any other class to get a different array list type

Declaring and Using ArrayLists

- `ArrayList<String>` is first constructed, it has size 0
- Use the `add` method to add an object to the end of the array list:

```
names.add("Emily"); // Now names has size 1 and element "Emily"
names.add("Bob"); // Now names has size 2 and elements "Emily", "Bob"
names.add("Cindy"); // names has size 3 and elements "Emily", "Bob", and "Cindy"
```

- The `size` method gives the current size of the array list.

  Size is now 3

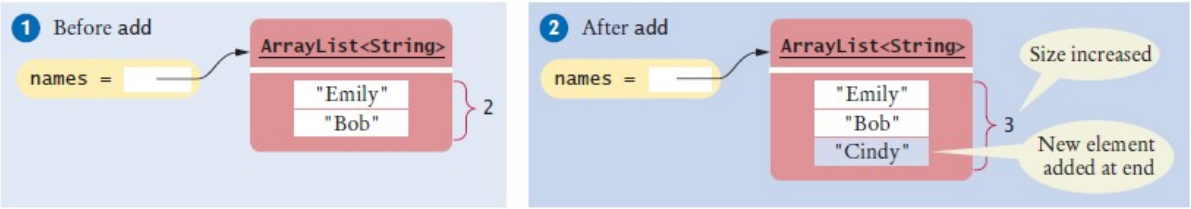


Figure 17 Adding an Array List Element with `add`

Declaring and Using ArrayLists

- To obtain an array list element, use the `get` method

  Index starts at 0

- To retrieve the name with index 2:

```
String name = names.get(2); // Gets the third element of the array list
```

- The last valid index is `names.size() - 1`

  A common bounds error:

```
int i = names.size();
name = names.get(i); // Error
```

- To set an array list element to a new value, use the `set` method:

```
names.set(2, "Carolyn");
```

Declaring and Using ArrayLists

- An array list has methods for adding and removing elements in the middle.

- This statement adds a new element at position 1 and moves all elements with index 1 or larger by one position.

```
names.add(1, "Ann")
```

- The remove method,

  removes the element at a given position
  moves all elements after the removed element down by one position
  and reduces the size of the array list by 1.

```
names.remove(1);
```

- To print an array list:

```
System.out.println(names); // Prints [Emily, Bob, Carolyn]
```

Declaring and Using ArrayLists

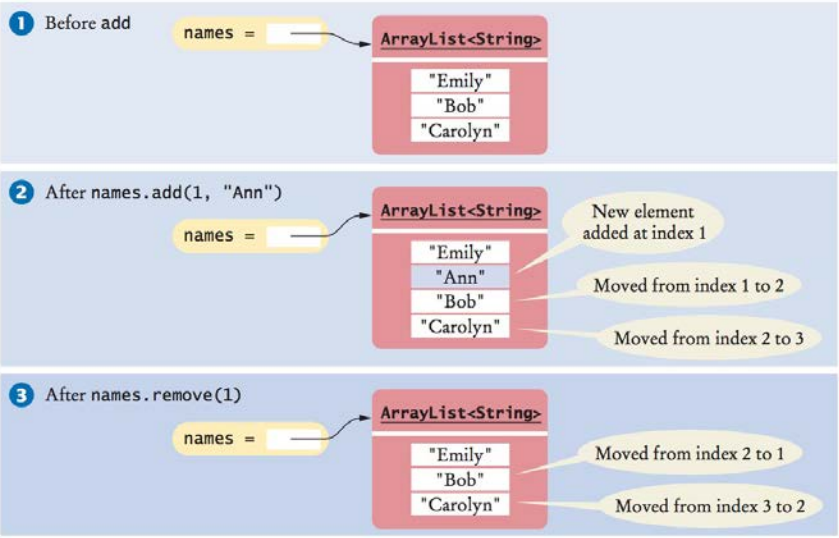


Figure 18 Adding and Removing Elements in the Middle of an Array List

Using the Enhanced for Loop with Array Lists

- You can use the enhanced for loop to visit all the elements of an array list

```
ArrayList<String> names = . . . ;
for (String name : names)
{
  System.out.println(name);
}
```

- This is equivalent to:

```
for (int i = 0; i < names.size(); i++)
{
  String name = names.get(i);
  System.out.println(name);
}
```

Copying Array Lists

- Copying an array list reference yields two references to the same array list.
- After the code below is executed

Both `names` and `friends` reference the same array list to which the string `"Harry"` was added.

```
ArrayList<String> friends = names;
friends.add("Harry");
```

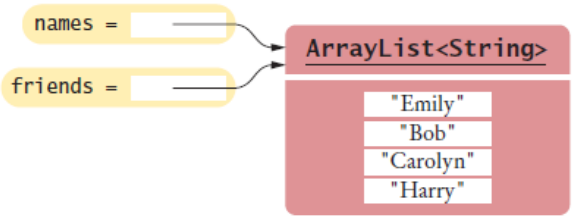


Figure 19 Copying an Array List Reference

- To make a copy of an array list, construct the copy and pass the original list into the constructor:

```
ArrayList<String> newNames = new ArrayList<String>(names);
```

Working with Array Lists

```
ArrayList<String> names =
  new ArrayList<String>();
```

Constructs an empty array list that can hold strings.

```
names.add("Ann");
names.add("Cindy");
```

Adds elements to the end.

```
System.out.println(names);
```

Prints `[Ann, Cindy]`.

```
names.add(1, "Bob");
```

Inserts an element at index 1. `names` is now `[Ann, Bob, Cindy]`.

```
names.remove(0);
```

Removes the element at index 0. `names` is now `[Bob, Cindy]`.

```
names.set(0, "Bill");
```

Replaces an element with a different value. `names` is now `[Bill, Cindy]`.

```
String name = names.get(i);
```

Gets an element.

```
String last =
  names.get(names.size() - 1);
```

Gets the last element.

```
  ArrayList<Integer> squares =
     new ArrayList<Integer>();
  for (int i = 0; i < 10; i++)
{
  squares.add(i * i);
}
```

Constructs an array list holding the first ten squares.

Wrapper Classes

- You cannot directly insert primitive type values into array lists.
- Like truffles that must be in a wrapper to be sold, a number must be placed in a wrapper to be stored in an array list.

© sandoclr/iStockphoto.

- Use the matching wrapper class.

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

Wrapper Classes

- To collect `double` values in an array list, you use an `ArrayList<Double>`.
- if you assign a `double` value to a `Double` variable, the number is automatically "put into a box"
- Called **auto-boxing:**

Automatic conversion between primitive types and the corresponding wrapper classes:

```
Double wrapper = 29.95;
```

Wrapper values are automatically "unboxed" to primitive types

```
double x = wrapper;
```

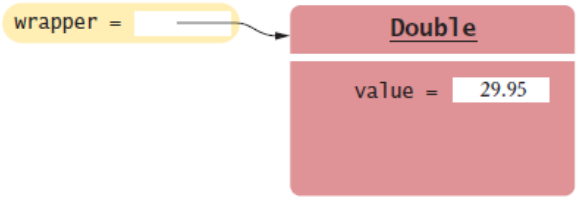


Figure 20 A Wrapper Class Variable

Using Array Algorithmswith Array Lists

- The array algorithms can be converted to array lists simply by using the array list methods instead of the array syntax.
- Code to find the largest element in an **array**:

```
double largest = values[0];
for (int i = 1; i < values.length; i++)
{
  if (values[i] > largest)
  {
     largest = values[i];
  }
}
```

- Code to find the largest element in an **array list**

```
double largest = values.get(0);
for (int i = 1; i < values.size(); i++)
{
  if (values.get(i) > largest)
  {
     largest = values.get(i);
  }
}
```

Storing Input Values in an ArrayList

- To collect an unknown number of inputs, array lists are much easier to use than arrays.
- Simply read the inputs and add them to an array list:

```
ArrayList<Double> inputs =
new ArrayList<Double>();
while (in.hasNextDouble())
{
  inputs.add(in.nextDouble());
}
```

Removing Matches

- To remove elements from an array list, call the `remove` method.
- Error: skips the element after the moved element

```
ArrayList<String> words = ...;
for (int i = 0; i < words.size(); i++)
{
  String word = words.get(i);
  if (word.length() < 4)
  {
     Remove the element at index i.
  }
}
```

- Concrete example

i	words
~~0~~	~~"Welcome", "to", "the", "island"~~
~~1~~	"Welcome", "the", "island"
2	

- Should not increment `i` when an element is removed

Removing Matches

- Pseudocode

```
If the element at index i matches the condition
Remove the element.
Else  Increment i.
```

- Use a `while` loop, not a `for` loop

```
int i = 0;
while (i < words.size())
{
  String word = words.get(i);
  if (word.length() < 4)
  {
    words.remove(i);
  }
  else
  {
    i++;
  }
}
```

Choosing Between Array Lists and Arrays

- For most programming tasks, array lists are easier to use than arrays

  Array lists can grow and shrink.

  Arrays have a nicer syntax.

- Recommendations

  If the size of a collection never changes, use an array.

  If you collect a long sequence of primitive type values and you are concerned about efficiency, use an array.

  Otherwise, use an array list.

Choosing Between Array Lists and Arrays

Table 3 Comparing Array and Array List Operations

Operation	Arrays	Array Lists
Get an element.	`x = values[4];`	`x = values.get(4);`
Replace an element.	`values[4] = 35;`	`values.set(4, 35);`
Number of elements.	`values.length`	`values.size()`
Number of filled elements.	`currentSize` (companion variable, see Section 7.1.4)	`values.size()`
Remove an element.	See Section 7.3.6.	`values.remove(4);`
Add an element, growing the collection.	See Section 7.3.7.	`values.add(35);`
Initializing a collection.	`int[] values = { 1, 4, 9 };`	No initializer list syntax; call `add` three times.

section_7/LargestInArrayList.java

```
1  import java.util.ArrayList;
2  import java.util.Scanner;
3
4  /**
5     This program reads a sequence of values and prints them, marking the largest value.
6  */
7  public class LargestInArrayList
8  {
9     public static void main(String[] args)
10    {
11       ArrayList<Double> values = new ArrayList<Double>();
12
13       // Read inputs
14
15       System.out.println("Please enter values, Q to
16       quit:");  Scanner in = new Scanner(System.in);
17       while (in.hasNextDouble())
18       {
19          values.add(in.nextDouble());
20       }
21
22       // Find the largest value
23
24       double largest = values.get(0);
25       for (int i = 1; i < values.size(); i++)
26       {
27          if (values.get(i) > largest)
28          {
29             largest = values.get(i);
30          }
31       }
32
33       // Print all values, marking the largest
34
35       for (double element : values)
```

Program Run:

```
Please enter values, Q to quit:
35 80 115 44.5 Q
35 80
115 <== largest value
44.5
```

Regression Testing

- **Test suite:** a set of tests for repeated testing
- **Cycling:** bug that is fixed but reappears in later versions
- **Regression testing:** involves repeating previously run tests to ensure that known failures of prior versions do not appear in new versions

Regression Testing - Two Approaches

- Organize a suite of test with multiple tester classes: `ScoreTester1`, `ScoreTester2`, ...

```
public class ScoreTester1
{
  public static void main(String[] args)
  {
    Student fred = new
    Student(100);
    fred.addScore(10);
    fred.addScore(20);
    fred.addScore(5);
    System.out.println("Final score: " +
    fred.finalScore());
    System.out.println("Expected: 30");
  }
}
```

- Provide a generic tester, and feed it inputs from multiple files.

section_8/ScoreTester.java

Generic tester:

```
1   import java.util.Scanner;
2
3   public class ScoreTester
4   {
5     public static void main(String[] args)
6     {
7       Scanner in = new Scanner(System.in);
8       double expected = in.nextDouble();
9       Student fred = new Student(100);
10      while (in.hasNextDouble())
11      {
12        if (!fred.addScore(in.nextDouble()))
13        {
14          System.out.println("Too many scores.");
15          return;
16        }
17      }
18      System.out.println("Final score: " + fred.finalScore());
19      System.out.println("Expected: " + expected);
20    }
21  }
```

Input and Output Redirection

Section_8/input1.txt

```
30
10
20
5
```

- Type the following command into a shell window

Input redirection

```
java ScoreTester < input1.txt
```

- Program Run:

```
Final score: 30
Expected: 30
```

- Output redirection:

```
java ScoreTester < input1.txt > output1.txt
```