### **Array Lists and Arrays**

## **Chapter Goals**

* To become familiar with using array lists to collect objects
* To learn about common array algorithms
* To be able to use arrays
* To understand when to choose array lists and arrays in your programs
* To implement partially filled arrays
* To learn how to use two-dimensional arrays

## **Array Lists**

* Consider Purse class
* Purse doesn't remember individual Coin objects, just the total
* Don't know how many coins--can't have variables coin1...coin10
* Use ArrayList to store variable number of objects  
  ArrayList coins = new ArrayList();  
  coins.add(new Coin(0.1, "dime"));  
  . . .
* size method yields number of elements

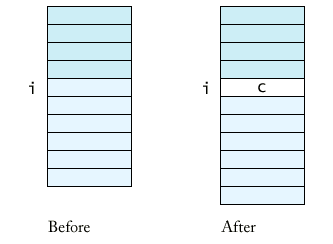
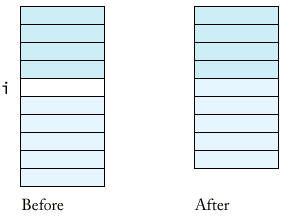
## **Retrieving Array List Elements**

* Use get method
* Index starts at 0
* Must cast to correct type
* Coin c = coins.get(0); // gets first coin
* Bounds error if index is out of range
* Most common bounds error:  
  int n = coins.size();  
  c = (Coin)coins.get(n); // ERROR   
  // legal index values are 0...n-1

## **Stepping Through all Elements**

for (int i = 0; i < coins.size(); i++)  
{  
 Coin c = (Coin)coins.get(i);  
 *do something with* c  
}

## **Adding and Removing Elements**

* set overwrites an existing value  
  coins.set(4, aNickel);
* add adds a new value before the index  
  add(i, c)  
  
* remove removes an element at an index  
       

## **File Purse.java**

| 1 | import java.util.ArrayList; |
| --- | --- |
| 2 |  |
| 3 | /\*\* |
| 4 | A purse holds a collection of coins. |
| 5 | \*/ |
| 6 | public class Purse |
| 7 | { |
| 8 | /\*\* |
| 9 | Constructs an empty purse. |
| 10 | \*/ |
| 11 | public Purse() |
| 12 | { |
| 13 | coins = new ArrayList(); |
| 14 | } |
| 15 |  |
| 16 | /\*\* |
| 17 | Add a coin to the purse. |
| 18 | @param aCoin the coin to add |
| 19 | \*/ |
| 20 | public void add(Coin aCoin) |
| 21 | { |
| 22 | coins.add(aCoin); |
| 23 | } |
| 24 |  |
| 25 | /\*\* |
| 26 | Get the total value of the coins in the purse. |
| 27 | @return the sum of all coin values |
| 28 | \*/ |
| 29 | public double getTotal() |
| 30 | { |
| 31 | double total = 0; |
| 32 | for (int i = 0; i < coins.size(); i++) |
| 33 | { |
| 34 | Coin aCoin = (Coin)coins.get(i); |
| 35 | total = total + aCoin.getValue(); |
| 36 | } |
| 37 | return total; |
| 38 | } |
| 39 |  |
| 40 | private ArrayList coins; |
| 41 | } |
| 42 |  |

## **Linear Search Algorithm**

public class Purse   
{   
 public boolean find(Coin aCoin)   
 {   
 for (int i = 0; i < coins.size(); i++)   
 {   
 Coin c =(Coin)coins.get(i);   
 if (c.equals(aCoin)) return true; //found a match   
 }   
 return false; //no match in the entire array list   
 }   
 ...   
}

## **Counting**

public class Purse   
{   
 public int count(Coin aCoin)   
 {   
 int matches = 0;   
 for (int i = 0; i < coins.size(); i++)   
 {  
 Coin c =(Coin)coins.get(i);   
 if (c.equals(aCoin)) matches++; //found a match   
 }   
 return matches;   
 }   
 ...   
}

## **Finding Maximum**

public class Purse   
{   
 public Coin getMaximum()   
 {   
 Coin max =(Coin)coins.get(0);   
 for (int i = **1**; i <coins.size(); i++) // loop starts at 1  
  {   
 Coin c =(Coin)coins.get(i);   
 if (c.getValue()>max.getValue()) max =c;   
 }   
 return max;   
 }   
 ...   
}

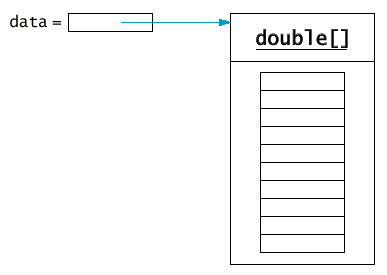
## **File Purse.java**

| 1 | import java.util.ArrayList; |
| --- | --- |
| 2 |  |
| 3 | /\*\* |
| 4 | A purse holds a collection of coins. |
| 5 | \*/ |
| 6 | public class Purse |
| 7 | { |
| 8 | /\*\* |
| 9 | Constructs an empty purse. |
| 10 | \*/ |
| 11 | public Purse() |
| 12 | { |
| 13 | coins = new ArrayList(); |
| 14 | } |
| 15 |  |
| 16 | /\*\* |
| 17 | Add a coin to the purse. |
| 18 | @param aCoin the coin to add |
| 19 | \*/ |
| 20 | public void add(Coin aCoin) |
| 21 | { |
| 22 | coins.add(aCoin); |
| 23 | } |
| 24 |  |
| 25 | /\*\* |
| 26 | Get the total value of the coins in the purse. |
| 27 | @return the sum of all coin values |
| 28 | \*/ |
| 29 | public double getTotal() |
| 30 | { |
| 31 | double total = 0; |
| 32 | for (int i = 0; i < coins.size(); i++) |
| 33 | { |
| 34 | Coin aCoin = (Coin)coins.get(i); |
| 35 | total = total + aCoin.getValue(); |
| 36 | } |
| 37 | return total; |
| 38 | } |
| 39 |  |
| 40 | /\*\* |
| 41 | Counts the number of coins in the purse |
| 42 | @return the number of coins |
| 43 | \*/ |
| 44 | public int count() |
| 45 | { |
| 46 | return coins.size(); |
| 47 | } |
| 48 |  |
| 49 | /\*\* |
| 50 | Tests if the purse has a coin that matches |
| 51 | a given coin. |
| 52 | @param aCoin the coin to match |
| 53 | @return true if there is a coin equal to aCoin |
| 54 | \*/ |
| 55 | public boolean find(Coin aCoin) |
| 56 | { |
| 57 | for (int i = 0; i < coins.size(); i++) |
| 58 | { |
| 59 | Coin c = (Coin)coins.get(i); |
| 60 | if (c.equals(aCoin)) return true; // found a match |
| 61 | } |
| 62 | return false; // no match in the entire array list |
| 63 | } |
| 64 |  |
| 65 | /\*\* |
| 66 | Counts the number of coins in the purse that match |
| 67 | a given coin. |
| 68 | @param aCoin the coin to match |
| 69 | @return the number of coins equal to aCoin |
| 70 | \*/ |
| 71 | public int count(Coin aCoin) |
| 72 | { |
| 73 | int matches = 0; |
| 74 | for (int i = 0; i < coins.size(); i++) |
| 75 | { |
| 76 | Coin c = (Coin)coins.get(i); |
| 77 | if (c.equals(aCoin)) matches++; // found a match |
| 78 | } |
| 79 | return matches; |
| 80 | } |
| 81 |  |
| 82 | /\*\* |
| 83 | Finds the coin with the largest value. |
| 84 | (Precondition: The purse is not empty) |
| 85 | @return a coin with maximum value in this purse |
| 86 | \*/ |
| 87 | Coin getMaximum() |
| 88 | { |
| 89 | Coin max = (Coin)coins.get(0); |
| 90 | for (int i = 1; i < coins.size(); i++) |
| 91 | { |
| 92 | Coin c = (Coin)coins.get(i); |
| 93 | if (c.getValue() > max.getValue()) |
| 94 | max = c; |
| 95 | } |
| 96 | return max; |
| 97 | } |
| 98 |  |
| 99 | private ArrayList coins; |
| 100 | } |
| 101 |  |

## **Storing Numbers in an Array List**

* Array lists store objects
* Use wrapper classes to store numbers
* Double wrapper = new Double(29.95);  
  double unwrapped = wrapper.doubleValue()
* ArrayList data = new ArrayList();  
  data.add(wrapper);  
  unwrapped = ((Double).data.get(i)).doubleValue();
* Also have  Integer and Boolean wrappers

## **Arrays**

* Construct array:  
  new double[10]
* Store in variable of type double[]  
  double[] data = new double[10];  
  
* Arrays have *fixed length*
* Arrays have element of specific type, not Object
* Use [] to access element:  
  data[4] = 29.95;
* Get array length as data.length. (Not a method!)

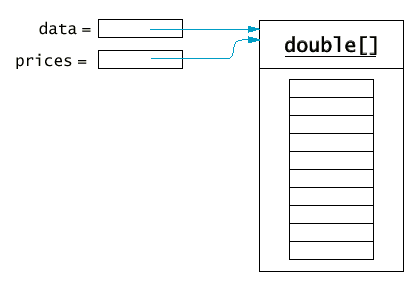
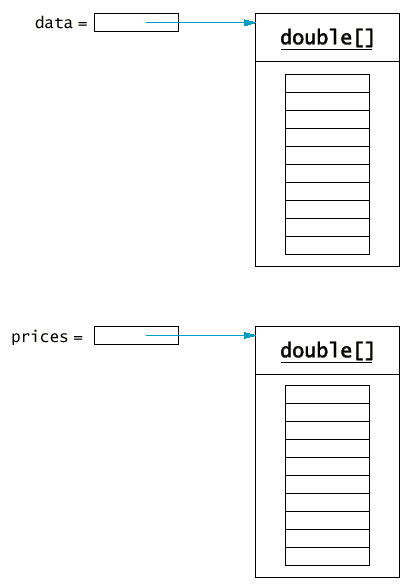
## **Syntax 13.1: Array Construction**

| |  | new *typename*[*length*] | | --- | --- |  **Example:**  |  | new double[10] | | --- | --- |  **Purpose:** To construct an array with a given number of elements. |
| --- | --- | --- | --- | --- |

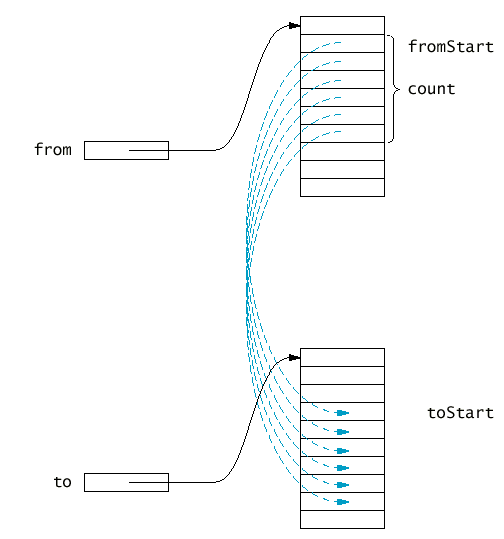
## **Syntax 13.2: Array Element Access**

| |  | *arrayReference*[*index*] | | --- | --- |  **Example:**  |  | a[4] = 29.95; double x = a[4]; | | --- | --- |  **Purpose:** To access an element in an array |
| --- | --- | --- | --- | --- |

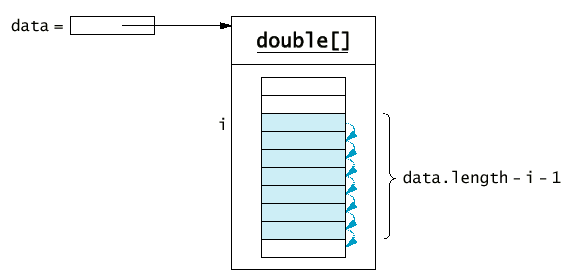
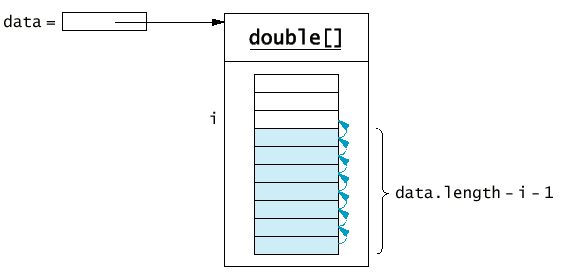
## **Copying Arrays**

* Copying an array reference yields a second reference to the same array  
  double[] data = new double[10];  
  // fill array . . .  
  double[] prices = data;  
  
* Use clone to make true copy  
  double[] prices = (double[])data.clone();  
  

## **Copying Array Elements**

System.arraycopy(from, fromStart, to, toStart, count);  


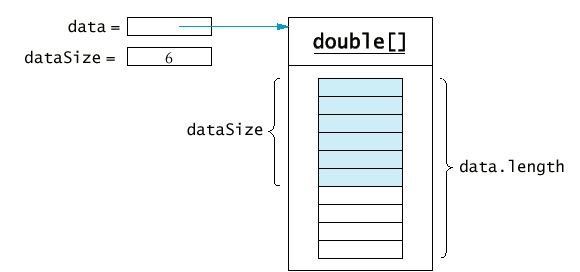
## **Adding and Removing Array Elements**

* Add an element:  
  System.arraycopy(data, i, data, i + 1, data.length - i - 1);  
  data[i] = x;  
  
* Remove an element:  
  System.arraycopy(data, i + 1, data, i, data.length - i - 1);  
  

## **Partially Filled Arrays**

* Array length = maximum number of elements in array
* Usually, array is partially filled
* Need companion variable to keep track of current size
* Uniform naming convention:  
  final int DATA**\_LENGTH** = 100;  
  double[] data = new double[DATA\_LENGTH];  
  int data**Size** = 0;
* Update dataSize as array is filled:  
  data[dataSize] = x;  
  dataSize++;

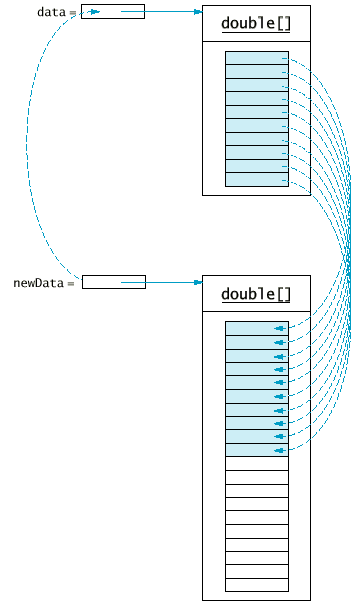
## **Partially Filled Arrays**



## **Partially Filled Arrays**

* Remember to stop at dataSize when looking at array elements:  
  for (int i = 0; i < dataSize; i++)  
     sum = sum + data[i];
* Be careful not to overfill the array  
  if (dataSize >= data.length)  
     System.out.println("Sorry--array full");
* Or grow the array:  
  double newData = new double[2 \* data.length];  
  System.arraycopy(data, 0, newData, 0, data.length);  
  data = newData;

## **Growing an Array**



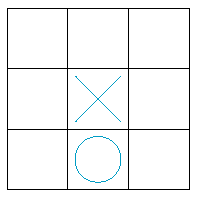
## **File DataSet.java**

| 1 | /\*\* |
| --- | --- |
| 2 | This class computes the average of a set of data values. |
| 3 | \*/ |
| 4 | public class DataSet |
| 5 | { |
| 6 | /\*\* |
| 7 | Constructs an empty data set. |
| 8 | \*/ |
| 9 | public DataSet() |
| 10 | { |
| 11 | final int DATA\_LENGTH = 100; |
| 12 | data = new double[DATA\_LENGTH]; |
| 13 | dataSize = 0; |
| 14 | } |
| 15 |  |
| 16 | /\*\* |
| 17 | Adds a data value to the data set |
| 18 | @param x a data value |
| 19 | \*/ |
| 20 | public void add(double x) |
| 21 | { |
| 22 | if (dataSize >= data.length) |
| 23 | { |
| 24 | // make a new array of twice the size |
| 25 | double[] newData = new double[2 \* data.length]; |
| 26 | // copy over all elements from data to newData |
| 27 | System.arraycopy(data, 0, newData, 0, data.length); |
| 28 | // abandon the old array and store in data |
| 29 | // a reference to the new array |
| 30 | data = newData; |
| 31 | } |
| 32 | data[dataSize] = x; |
| 33 | dataSize++; |
| 34 | } |
| 35 |  |
| 36 | /\*\* |
| 37 | Gets the average of the added data. |
| 38 | @return the average or 0 if no data has been added |
| 39 | \*/ |
| 40 | public double getAverage() |
| 41 | { |
| 42 | if (dataSize == 0) return 0; |
| 43 | double sum = 0; |
| 44 | for (int i = 0; i < dataSize; i++) |
| 45 | sum = sum + data[i]; |
| 46 | return sum / dataSize; |
| 47 | } |
| 48 |  |
| 49 | private double[] data; |
| 50 | private int dataSize; |
| 51 | } |

## **File DataSetTest.java**

| 1 | import java.util.Random; |
| --- | --- |
| 2 |  |
| 3 | /\*\* |
| 4 | This program tests the DataSet class by adding 10,000 numbers |
| 5 | to the data set and computing the average. |
| 6 | \*/ |
| 7 | public class DataSetTest |
| 8 | { |
| 9 | public static void main(String[] args) |
| 10 | { |
| 11 | Random generator = new Random(); |
| 12 | DataSet data = new DataSet(); |
| 13 | final int COUNT = 10000; |
| 14 | System.out.println("Adding " + COUNT + " random numbers."); |
| 15 | for (int i = 0; i < COUNT; i++) |
| 16 | { |
| 17 | double x = generator.nextDouble(); |
| 18 | data.add(x); |
| 19 | } |
| 20 | double average = data.getAverage(); |
| 21 | System.out.println("average=" + average); |
| 22 | } |
| 23 | } |

## **Two-Dimensional Arrays**

* Matrix with rows and columns
* Example: Tic Tac Toe board  
  
* char[][] board = new char[3][3];  
  board[i][j] = 'x';

## **File TicTacToe.java**

| 1 | /\*\* |
| --- | --- |
| 2 | A 3 x 3 Tic-Tac-Toe board. |
| 3 | \*/ |
| 4 | public class TicTacToe |
| 5 | { |
| 6 | /\*\* |
| 7 | Constructs an empty board. |
| 8 | \*/ |
| 9 | public TicTacToe() |
| 10 | { |
| 11 | board = new char[ROWS][COLUMNS]; |
| 12 |  |
| 13 | // fill with spaces |
| 14 | for (int i = 0; i < ROWS; i++) |
| 15 | for (int j = 0; j < COLUMNS; j++) |
| 16 | board[i][j] = ' '; |
| 17 | } |
| 18 |  |
| 19 | /\*\* |
| 20 | Sets a field in the board. The field must be unoccupied. |
| 21 | @param i the row index |
| 22 | @param j the column index |
| 23 | @param player the player ('x' or 'o') |
| 24 | \*/ |
| 25 | public void set(int i, int j, char player) |
| 26 | { |
| 27 | if (board[i][j] != ' ') |
| 28 | throw new IllegalArgumentException("Position occupied"); |
| 29 | board[i][j] = player; |
| 30 | } |
| 31 |  |
| 32 | /\*\* |
| 33 | Creates a string representation of the board such as |
| 34 | |x  o| |
| 35 | |  x | |
| 36 | |   o| |
| 37 | @return the string representation |
| 38 | public String toString() |
| 39 | { |
| 40 | String r = ""; |
| 41 | for (int i = 0; i < ROWS; i++) |
| 42 | { |
| 43 | r = r + "|"; |
| 44 | for (int j = 0; j < COLUMNS; j++) |
| 45 | r = r + board[i][j]; |
| 46 | r = r + "|\n"; |
| 47 | } |
| 48 | return r; |
| 49 | } |
| 50 |  |
| 51 | private char[][] board; |
| 52 | private static final int ROWS = 3; |
| 53 | private static final int COLUMNS = 3; |
| 54 | } |

## **File TicTacToeTest.java**

| 1 | import javax.swing.JOptionPane; |
| --- | --- |
| 2 |  |
| 3 | /\*\* |
| 4 | This program tests the TicTacToe class by prompting the |
| 5 | user to set positions on the board and printing out the |
| 6 | result. |
| 7 | \*/ |
| 8 | public class TicTacToeTest |
| 9 | { |
| 10 | public static void main(String[] args) |
| 11 | { |
| 12 | char player = 'x'; |
| 13 | TicTacToe game = new TicTacToe(); |
| 14 | while (true) |
| 15 | { |
| 16 | System.out.println(game); // calls game.toString() |
| 17 | String input = JOptionPane.showInputDialog( |
| 18 | "Row for " + player + " (Cancel to exit)"); |
| 19 | if (input == null) System.exit(0); |
| 20 | int row = Integer.parseInt(input); |
| 21 | input = JOptionPane.showInputDialog( |
| 22 | "Column for " + player); |
| 23 | int column = Integer.parseInt(input); |
| 24 | game.set(row, column, player); |
| 25 | if (player == 'x') player = 'o'; else player = 'x'; |
| 26 | } |
| 27 | } |
| 28 | } |