SINGLE-DIMENSIONAL ARRAYS

Array

- *single array variable can reference a large collection of data*
- stores a fixed-size sequential collection of elements of the same type
- *Once an array is created, its size is fixed*
- *array reference variable is used to access the elements in an array using an* index
- Instead of declaring individual variables, such as **number0**, **number1**, . . . , and **number99**, declare one array variable such as **numbers** and use **numbers[0]**, **numbers[1]**, . . . , and **numbers[99]**

Declaring Array Variables

- declare a variable to reference the array and specify the array's *element type*
- Syntax: elementType[] arrayRefVar;
- use **elementType arrayRefVar[]** to declare an array variable
  - **elementType** can be any data type
  - **double**[] myList;

Creating Arrays

- declaration of an array variable does not allocate any space in memory for the array
- If a variable does not contain a reference to an array, the value of the variable is **null**
- cannot assign elements to an array unless it has already been created
- After an array variable is declared, you can create an array by using the **new** operator
  - arrayRefVar = **new** elementType[arraySize];
- (1) it creates an array using **new elementType[arraySize];**
- (2) it assigns the reference of the newly created array to the variable **arrayRefVar**.

- Combine in one statement
  - elementType[] arrayRefVar = new elementType[arraySize];   OR
  - elementType arrayRefVar[] = new elementType[arraySize];
  - **double**[] myList = **new double**[10];
- To assign values to the elements:
  - arrayRefVar[index] = value;
  - myList[0] = **5.6**;
  - myList[1] = **4.5**;
  - myList[2] = **3.3**;

Array Size and Default Values

- When space for an array is allocated, the array size must be given
- size of an array cannot be changed after the array is created
- Size can be obtained using **arrayRefVar.length**
- When an array is created, its elements are assigned the default value
  - 0 for the numeric primitive data types
  - \u0000 for char types
  - false for boolean types

Accessing Array Elements

- array elements are accessed through the index
- range from **0** to **arrayRefVar.length-1**
- Each element in the array is represented using syntax arrayRefVar[index];
- indexed variable can be used in the same way as a regular variable
  - myList[**2**] = myList[**0**] + myList[**1**];

Array Initializers

- combines the declaration, creation, and initialization of an array in one statement
  - elementType[] arrayRefVar = {value0, value1, ..., value*k*};
  - double[] myList = {1.9, 2.9, 3.4, 3.5};
- new operator is not used in the array-initializer syntax
- double[] myList;

  myList = {1.9, 2.9, 3.4, 3.5};

Processing Arrays

- When processing array elements, you will often use a **for** loop
- *Initializing arrays with input values*
- *Initializing arrays with random values*
- *Displaying arrays*
- *Summing all elements*
- *Finding the largest element*
- *Finding the smallest index of the largest element*
- *Shifting elements*
- **char**[] city = {'**D**', '**a**', '**l**', '**l**', '**a**', '**s**'};

  System.out.println(city);

Foreach Loops

- enables you to traverse the array sequentially without using an index variable
- **for** (**double** e: myList) {

  System.out.println(e);

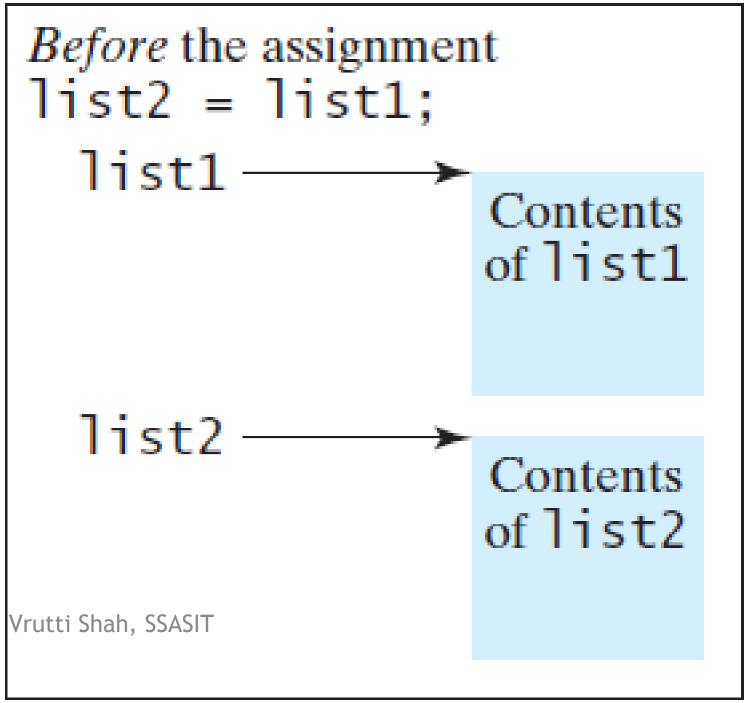
  }
- variable, **e**, must be declared as the same type as the elements in **myList**.
- Syntax:
- **for** (elementType element: arrayRefVar) {

  // Process the element

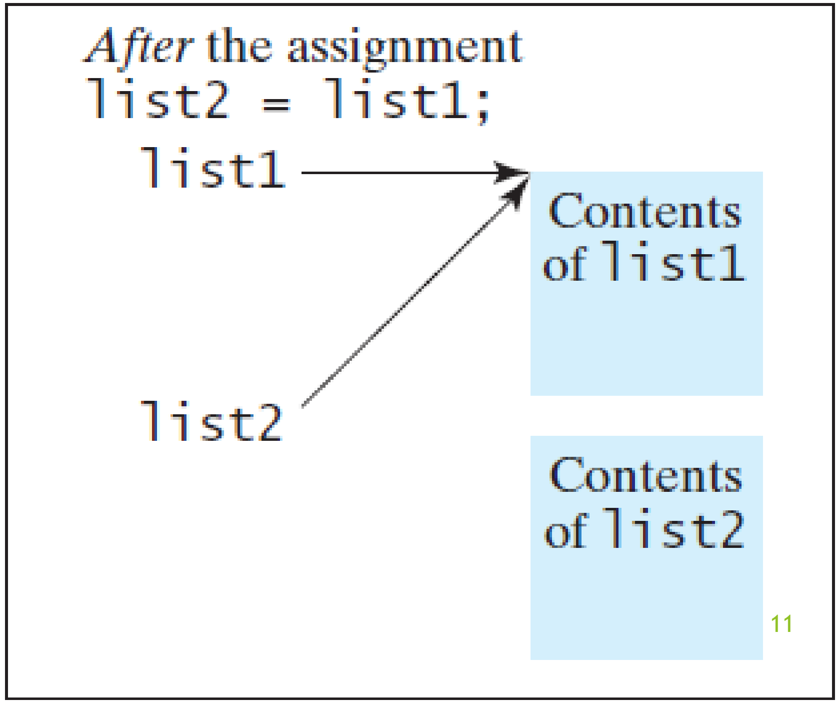
  }

Copying Arrays

- *To copy the contents of one array into another, you have to copy the array's individual elements into the other array.*
- list2 = list1;
- does not copy the contents of the array referenced by **list1** to **list2**
- **list1** and **list2** reference the same array



Before the assignment
`list2 = list1;`



After the assignment
`list2 = list1;`

Copying Arrays (Contd…)

- three ways to copy arrays
  - Use a loop to copy individual elements one by one
  - Use the static **arraycopy** method in the **System** class
  - Use the **clone** method to copy arrays
- Syntax of arraycopy method:
  - arraycopy(sourceArray, srcPos, targetArray, tarPos, length);
  - **srcPos** and **tarPos** indicate the starting positions in **sourceArray** and **targetArray**, respectively
  - number of elements copied from **sourceArray** to **targetArray** is indicated by **length**
- System.arraycopy(sourceArray, **0**, targetArray, **0**, sourceArray.length);

Copying Arrays (Contd…)

- **arraycopy** method does not allocate memory space for the target array
- target array must have already been created with its memory space allocated
- After the copying takes place, **targetArray** and **sourceArray** have the same content but independent memory locations

Passing Arrays to Methods

- *When passing an array to a method, the reference of the array is passed to the method*
-
  ```
  public static void printArray(int[] array) {
   for (int i = 0; i < array.length; i++) {
   System.out.print(array[i] + " ");
   }
   }
  ```
- To invoke the method: printArray(**new int**[]{**3**, **1**, **2**, **6**, **4**, **2**});
- There is no explicit reference variable for the array. Such array is called an *anonymous array*

Passing Arrays to Methods (Contd…)

- Differences between passing the values of variables of primitive data types and passing arrays
  - For an argument of a primitive type, the argument’s value is passed
  - For an argument of an array type, the value of the argument is a reference to an array (pass-by-sharing),
  - array in the method is the same as the array being passed
  - if you change the array in the method, you will see the change outside the method

```
public class Test {
 public static void main(String[] args) {
 int x = 1; // x represents an int value
 int[] y = new int[10]; // y represents an array of int values
 m(x, y); // Invoke m with arguments x and y
 System.out.println("x is " + x);
 System.out.println("y[0] is " + y[0]);
 }
 public static void m(int number, int[] numbers) {
 number = 1001; // Assign a new value to number
 numbers[0] = 5555; // Assign a new value to numbers[0]
 }
 }
```

Returning an Array from a Method

- *When a method returns an array, the reference of the array is returned*

-
  ```
  public static int[] reverse(int[] list) {
    int[] result = new int[list.length];
    for (int i = 0, j = result.length - 1;
    i < list.length; i++, j--) {
    result[j] = list[i];
     }
    return result;
    }
  ```

-
  ```
  int[] list1 = {1, 2, 3, 4, 5, 6};
    int[] list2 = reverse(list1);
  ```

Variable-Length Argument Lists

- *A variable number of arguments of the same type can be passed to a method and treated as an array.*
- Syntax: typeName... parameterName
- In the method declaration, you specify the type followed by an ellipsis (...)
- Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter

Variable-Length Argument Lists (Contd…)

```
public class VarArgsDemo {
  public static void main(String[] args) {
  printMax(34, 3, 3, 2, 56.5);
  printMax(new double[]{1, 2, 3});
  }
  public static void printMax(double... numbers) {
  if (numbers.length == 0) {
  System.out.println("No argument passed");
  return;
  }
```

The Arrays Class

- *contains useful methods for common array operations such as sorting and searching*
- java.util.Arrays class contains various static methods for
  - sorting and searching arrays
  - comparing arrays
  - filling array elements
  - returning a string representation of the array
- sort or parallelSort method to sort a whole array or a partial array
  - java.util.Arrays.sort(numbers);
  - java.util.Arrays.sort(chars, **1**, **3**);

The Arrays Class (Contd…)

- **binarySearch** method to search for a key in an array
- array must be presorted in increasing order
- If the key is not in the array, the method returns **-(insertionIndex + 1)**
  - java.util.Arrays.binarySearch(list, **11**));
- use the **equals** method to check whether two arrays are strictly equal
  - System.out.println(java.util.Arrays.equals(list1, list2));
- use the **fill** method to fill in all or part of the array
  - java.util.Arrays.fill(list1, **5**);
  - java.util.Arrays.fill(list2, **1**, **5**, **8**);
- **toString** method to return a string that represents all elements in the array
  - System.out.println(Arrays.toString(list));