Arrays (7.1)

- **array**: An object that stores many values of the same type.
  - **element**: One value in an array.
  - **index**: A 0-based integer to access an element from an array.
Array declaration

\[ \text{type[]} \ \text{name} \ = \ \text{new} \ \text{type}[\text{length}]; \]

- Example:
  \[
  \text{int[]} \ \text{numbers} \ = \ \text{new} \ \text{int}[10];
  \]

- All elements' values are initially 0.

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Accessing elements

name[index] // access
name[index] = value; // modify

- Example:

```
numbers[0] = 27;
numbers[3] = -6;

System.out.println(numbers[0]);
if (numbers[3] < 0) {
    System.out.println("value 3 is negative");
}
```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>27</td>
<td>0</td>
<td>0</td>
<td>-6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Out-of-bounds

• Legal indexes: between 0 and the array's length - 1.
  – Reading or writing any index outside this range will throw an ArrayIndexOutOfBoundsException.

• Example:
  ```java
  int[] data = new int[10];
  System.out.println(data[0]);  // okay
  System.out.println(data[9]);  // okay
  System.out.println(data[-1]); // exception
  System.out.println(data[10]); // exception
  ```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The length field

An array's length field stores its number of elements.

```java
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}
// output: 0 2 4 6 8 10 12 14
```

- It does not use parentheses like a String's .length().
Quick initialization

\[ \text{type}[] \ \text{name} = \{ \text{value}, \ \text{value}, \ldots \ \text{value} \}; \]

- Example:
  ```
  int[] numbers = \{ 12, 49, -2, 26, 5, 17, -6 \};
  ```

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>12</td>
<td>49</td>
<td>-2</td>
<td>26</td>
<td>5</td>
<td>17</td>
<td>-6</td>
</tr>
</tbody>
</table>

- Useful when you know what the array's elements will be.
- The compiler figures out the size by counting the values.
The Arrays class

- Class Arrays in package java.util has useful static methods for manipulating arrays:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarySearch(array, value)</td>
<td>returns the index of the given value in a sorted array (&lt; 0 if not found)</td>
</tr>
<tr>
<td>copyOf(array, length)</td>
<td>returns a new array with same elements</td>
</tr>
<tr>
<td>equals(array1, array2)</td>
<td>returns true if the two arrays contain the same elements in the same order</td>
</tr>
<tr>
<td>fill(array, value)</td>
<td>sets every element in the array to have the given value</td>
</tr>
<tr>
<td>sort(array)</td>
<td>arranges the elements in the array into ascending order</td>
</tr>
<tr>
<td>toString(array)</td>
<td>returns a string representing the array, such as &quot;[10, 30, 17]&quot;</td>
</tr>
</tbody>
</table>
Array as parameter

public static type methodName(type[] name) {

  // Example:
  public static double average(int[] numbers) {
    ...
  }

  // Call:
  methodName(arrayName);

  // Example:
  int[] scores = {13, 17, 12, 15, 11};
  double avg = average(scores);
public static type[] methodName(parameters) {

– Example:

public static int[] countDigits(int n) {
    int[] counts = new int[10];
    ...
    return counts;
}

• Call:

    type[] name = methodName(parameters);

– Example:

    int[] tally = countDigits(229231007);
    System.out.println(Arrays.toString(tally));
Exercise

• Write a method named `stutter` that accepts an array of integers as a parameter and returns a new array, twice as long as the original, with two copies of each original element.

  – If the method were called in the following way:
    ```java
    int[] a = {4, 7, -2, 15, 6};
    int[] a2 = stutter(a);
    System.out.println(Arrays.toString(a2));
    ```

  – The output produced would be:
    ```java
    [4, 4, 7, 7, -2, -2, 15, 15, 6, 6]
    ```
public static int[] stutter(int[] a) {
    int[] result = new int[a.length * 2];
    for (int i = 0; i < a.length; i++) {
        result[2 * i] = a[i];
        result[2 * i + 1] = a[i];
    }
    return result;
}

public static int[] stutter(int[] a) {
    int[] result = new int[a.length * 2];
    for (int i = 0; i < result.length; i++) {
        result[i] = a[i / 2];
    }
    return result;
}
Testing code

- **Q**: How can we tell if our `stutter` method works properly?
  - **A**: We must test it.

- **Q**: How do we test code?
  - **A**: Call the method several times and print/examine the results.

- **Q**: Can we test all possible usages of this method?
  **Q**: Can we prove that the `stutter` code has no bugs?
  - **A**: No; exhaustive testing is impractical/impossible for most code.
  - **A**: No; testing finds bugs but cannot prove the absence of bugs.
• test case: Running a piece of code once on a given input.

• Q: Which cases should we choose to test?
  – equivalence classes of input: Think about kinds of inputs:
    • positive vs. negative numbers vs. 0; null (maybe)
    • unique values vs. duplicates (consecutive and non-consecutive)
    • an empty array; a 1-element array; a many-element array

• Q: What are some properties to look for in testing code?
  – boundaries: Hits cases close to a relevant boundary, e.g. the maximum allowed value, the first/last element in an array, etc.
  – code coverage: Hits all paths through code (if/elses, etc.)
  – preconditions: What does the method assume? Does the code ever violate those assumptions?
Exercise

• Write a short piece of code that tests the **stutter** method.
  – Decide on a group of test input cases.

  – For each test case:
    • Print the array's contents before and after stuttering.
    • Print whether the test was successful or failed.
public static void main(String[] args) {
    int[] a1 = {1, 2, 4, 5, 6};
    int[] a2 = stutter(a1);
    System.out.println(Arrays.toString(a2));
    ...
}

• Pros:
  – simple, short

• Cons:
  – must manually check output to see if it is correct
  – must copy/paste to create each test case (redundant)
public static void main(String[] args) {
    test(new int[] {1, 2, 4, 5, 6, 8},
         new int[] {1, 1, 2, 2, 4, 4, 5, 5, 6, 6, 8, 8});
    test(new int[] {0, 0, 7, 9},
         new int[] {0, 0, 0, 0, 7, 7, 9, 9});
    test(new int[] {-50, 95, -9876},
         new int[] {-50, -50, 95, 95, -9876, -9876});
    test(new int[] {42}, new int[] {42, 42});
    test(new int[] {}, new int[] {});
}

public static void test(int[] a, int[] expected) {
    int[] a2 = stutter(a);
    System.out.print(Arrays.toString(a) + " -> " +
                     Arrays.toString(a2) + " : ");
    if (Arrays.equals(a2, expected)) {
        System.out.println("Pass");
    } else {
        System.out.println("FAIL!!!");
    }
}