**Administrative**

**Today’s session**

Array

ArrayList

Array – ArrayList comparison

**Session Topics**

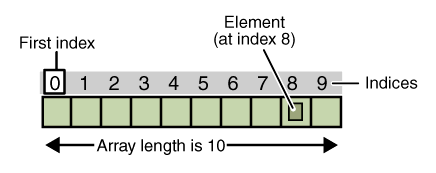
**Array**

● An **array** is a series of values of the same data type.

● Each value in an array is called an **element** or **member**.

● Each element in an array has an address called an **index** or **subscript**.

● Indices are numbered from zero to one less than the number of elements in the array.



**Array declaration**

● An **array declaration** reserves a spot in memory for an array.

● An array is of type object, not primitive.

● An array declaration has syntax:

<data type>[] <array-name> = new <data type>[<integer-expression>];

OR

<data type>[] <array-name> = {<value-list>};

OR

<data type>[] <array-name> = null;

OR

<data type>[] <array-name>;

**Where:**

**<data type>** is a primitive data type (byte, boolean, short, char, int, long, float, double) or class type (String, etc).

**<array-name>** has the same naming rules as a variable (begins with a letter, dollar sign, or the underscore, and is optionally followed by any sequence of letters, digits, dollar signs, or underscores).

**<integer-expression>** evaluates to a nonnegative integer.

● Array declaration examples:

int[] scores = new int[4];

double[] salaries = new double[100];

String[] daysOfWeek = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};

char[] vowels = {'a','e','i','o','u'};

boolean[] flags = new boolean[8];

**Array element reference**

● An **array element reference** uses an index to access an element within an array.

● An array element referencehas syntax:

<array-name>[<integer-expression>]

Where **<integer-expression>** mustevaluate to an integer between 0 and the array length minus one. An **index out-of-bounds error** results if the expression does not.

● An array element referencemay be used wherever an expression may appear. The data type of the referencemust be compatible the expression context.

● The array length is available through an instance variable called **<array-name>.length**

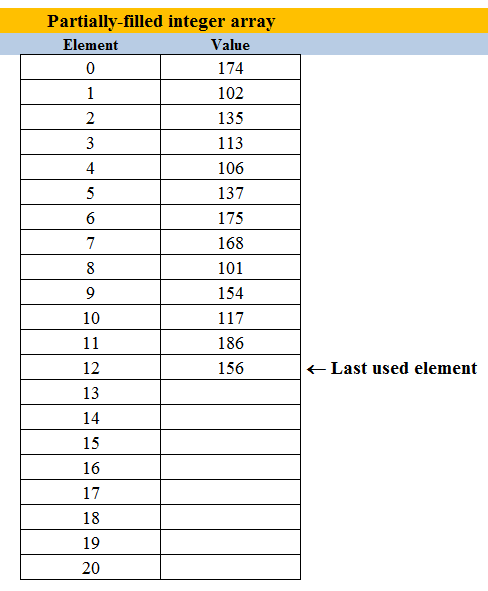
● See **Arrays** sample application on Blackboard.

**Partially-filled array**

● A **partially-filled array** is an array holding a number of values fewer than the array length.

● A partially-filled array requires an additional integer variable to keep track of the number of values stored in the array.

● Partially-filled array example:



**Array as parameter**

● A method may have an array as a parameter.

● When an array argument is matched to an array parameter:

✓ A pointer to the array is passed into the method, not any of its values.

✓ Any changes made to array element values within the method are also made to array element values outside the method.

● Method parameter and method call example:

public void someMethod(int[] arr)

{

…

}

public static void main (String[] args)

{

…

int[] numbers = new int[10];

…

someMethod(numbers);

…

}

**Array as returned value**

● A value method may return an array.

● A method call that returns an array may appear wherever an array of the same type may be used.

● Returned array example:

private int[] arrayWithValues(int elementCount)

{

int[] intArray = new int[elementCount];

…

(set elements in intArray array)

…

return intArray;

}

public static void main (String[] args)

{

…

int[] numbers = new int[10];

…

numbers = arrayWithValues(10);

…

}

● Several methods in java.util.Arrays are available for an array including:

|  |  |
| --- | --- |
| Method | Purpose |
| copyOf(<array>, <size>) | Create array from array <array> with size <size> and preserve the values from <array>. |
| fill(<array>, <value>) | Set each <array> element to <value>. |
| sort(<array>) | Sort <array> in ascending order (using Dual-Pivot Quicksort). |

● See **Arrays** sample application on Blackboard.

**Parallel arrays**

**● Parallel arrays** are two or more arrays of the same size defined to hold information in a database-like fashion.

**Multidimensional array**

● Enables the storage of data in multiple dimensions.

● A two-dimensional array declaration has syntax:

<data type>[][] <array-name> =

new <data type>[<integer-expression-1>][<integer-expression-2>];

OR

<data type>[][] <array-name> = {{<row1-value-list>}, {<row2-value-list>}};

OR

<data type>[][] <array-name> = null;

OR

<data type>[][] <array-name>;

**Where:**

**<data type>** is a primitive data type (byte, boolean, short, char, int, long, float, double) or class type (String, etc.).

**<array-name>** has the same naming rules as a variable (begins with a letter, dollar sign, or the underscore, and is optionally followed by any sequence of letters, digits, dollar signs, or underscores).

**<integer-expression>** evaluates to a nonnegative integer.

● In a multidimensional array, **<array-name>.length** refers to the length of the **first** dimension.

● In a two-dimensional array, **<array-name>[n].length** refers to the length of the **second** dimension.

**ArrayList**

● An **ArrayList** is similar to an array except that it can change size during program execution.

● Java actually stores an ArrayList as an array and recreates the array as needed.

● Like an array, each element in an ArrayList has the same base type.

● Unlike an array, the ArrayList base type must be a class. This means you cannot have an ArrayList of a primitive type (int, double, etc.). You can, however, have an ArrayList of a wrapper class.

● Here are the primitive data types and their corresponding wrapper classes:

|  |  |
| --- | --- |
| Primitive type | Wrapper class |
| boolean | Boolean |
| byte | Byte |
| char | Character |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |
| double | Double |

**ArrayList declaration**

● An **ArrayList declaration** reserves a spot in memory for an array list.

● An ArrayList is of type object, not primitive.

● An ArrayList declaration has syntax:

ArrayList<[data-type]> <arraylist-name> = new ArrayList<[data-type]>();

OR

ArrayList<[data-type]> <arraylist-name> = new ArrayList<[data-type]>(<initial-size>);

**Where:**

**[data type]** is a class type (Integer, Double, String, etc).

**<arraylist-name>** has the same naming rules as a variable (begins with a letter, dollar sign, or the underscore, and is optionally followed by any sequence of letters, digits, dollar signs, or underscores).

**<initial-size>** evaluates to a nonnegative integer.

● An ArrayList declared without an initial size has a capacity of ten.

● Although an ArrayList has an initial size, the number of accessible elements is zero until they are explicitly added.

● Several methods are available for an ArrayList including:

|  |  |
| --- | --- |
| Method | Purpose |
| add(<value>) | Add <value> to end of list. |
| clear() | Remove all values from list. |
| contains(<value>) | Search for <value> in list. |
| get(<index>) | Return value at <index>. |
| remove(<index>) | Remove value at <index>. |
| set(<index>, <value>) | Set value at <index> to <value>. |
| size() | Return number of values in list. |
| trimToSize() | Remove unused spots in list. |

● The for-each statement may be used to loop through ArrayList values.

● See **ArrayLists** sample application on Blackboard.

**Array – ArrayList comparison**

|  |  |
| --- | --- |
| Array | ArrayList |
| An array may be any primitive or reference data type. | An arraylist may only be a reference data type. |
| An array must be manually sized and resized. | An arraylist is automatically sized and resized. |
| Data gaps in an array must be manually closed. | Data gaps in an arraylist are automatically closed. |
| For larger arrays, search and sort operations are faster. | For larger arraylists, search and sort operations are slower. |
| For larger arrays, less memory is required. | For larger arraylists, more memory is required. |
| Some array operations must be manually coded. | Most Arraylist operations are already coded. |