Arrays

A problem with simple variables

- One variable holds one value
  - The value may change over time, but at any given time, a variable holds a single value
- If you want to keep track of many values, you need many variables
- All of these variables need to have names
- What if you need to keep track of hundreds or thousands of values?

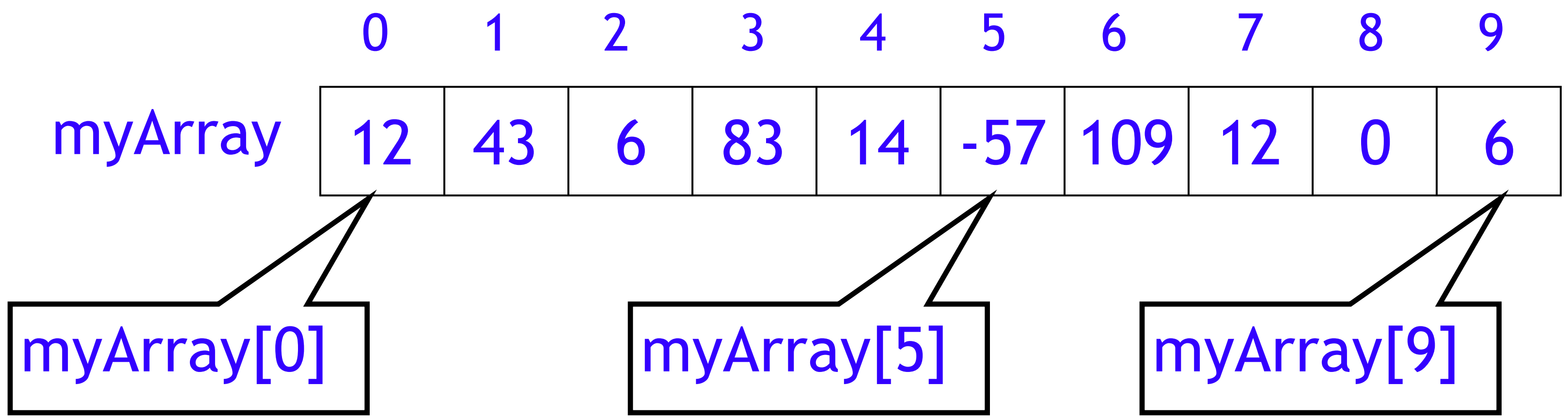
Multiple values

- An array lets you associate one name with a fixed (but possibly large) number of values
- All values must have the same type
- The values are distinguished by a numerical index between 0 and array size minus 1

	0	1	2	3	4	5	6	7	8	9
myArray	12	43	6	83	14	-57	109	12	0	6

Indexing into arrays

- To reference a single array element, use
  array-name [*index*]
- Indexed elements can be used just like simple variables
  - You can access their values
  - You can modify their values
- An array index is sometimes called a subscript



Using array elements

	0	1	2	3	4	5	6	7	8	9
myArray	12	43	6	83	14	-57	109	12	0	6

- Examples:

```
x = myArray[1];                // sets x to 43
myArray[4] = 99;               // replaces 14 with 99
m = 5;
y = myArray[m];                // sets y to -57
z = myArray[myArray[9]];       // sets z to 109
```

Array values

- An array may hold *any* type of value
- All values in an array must be the *same* type
    - For example, you can have:
        - an array of integers (ints)
        - an array of Strings
        - an array of Person
            - In this case, all the elements are Persons; but they may belong to different *subclasses* of Person
            - For example, if you have a class Employee extends Person, then you can put Employees in your array of Person
            - This is because an Employee **is a** Person
        - You can even have arrays of arrays, for example, an array of arrays of int

Strings and arrays

- Strings and arrays both have special syntax
- Strings are objects, and can be used as objects
- Arrays are objects, but
  - Arrays are created using special syntax:
    `new` *type*`[`*size*`]` instead of `new Person()`
  - If an array holds elements of type `T`, then the array's type is "array of `T`"

Declaration versus definition

- Arrays *are* objects
- Creating arrays is like creating other objects:
  - the *declaration* provides type information and allocates space for a *reference* to the array (when it is created)
  - the new *definition* actually allocates space for the array
  - declaration and definition may be separate or combined
- Example for ordinary objects:

```
Person p;                       // declaration
p = new Person("John");         // definition
Person p = new Person("John");  // combined
```

Declaring and defining arrays

- Example for array objects:
  - `int[ ] myArray;                // declaration`
    - This *declares* `myArray` to be an array of integers
    - It does not create an array—it only provides a place to put an array
    - Notice that the size is *not* part of the type
  - `myArray = new int[10];   // definition`
    - `new int[10]` creates the array
    - The rest is an ordinary assignment statement
  - `int[ ] myArray = new int[10];   // both`

Two ways to declare arrays

- You can declare more than one variable in the same declaration:

  ```
  int a[ ], b, c[ ], d;   // notice position of brackets
  ```

  - a and c are int arrays
  - b and d are just ints
- Another syntax:

  ```
  int [ ] a, b, c, d;      // notice position of brackets
  ```

  - a, b, c and d are int arrays
  - When the brackets come before the first variable, they apply to *all* variables in the list
- But...
  - In Java, we typically declare each variable separately

Array assignment

- Array assignment is object assignment
- Object assignment does *not* copy values

```
Person p1; Person p2;
p1 = new Person("John");
p2 = p1;  // p1 and p2 refer to the same person
```

- Array assignment does *not* copy values

```
int[ ] a1;  int[ ] a2;
a1 = new int[10];
a2 = a1; // a1 and a2 refer to the same array
```

An array’s size is *not* part of its type

- When you declare an array, you declare its type; you *must not* specify its size
  - Example: `String names[ ];`
- When you define the array, you allocate space; you *must* specify its size
  - Example: `names = new String[50];`
- This is true even when the two are combined
  - Example: `String names[ ] = new String[50];`

Array assignment

- When you assign an array value to an array variable, the types must be compatible
- The following is *not* legal:

```
double[ ] dub = new int[10]; // illegal
```

- The following *is* legal:

```
int[ ] myArray = new int[10];
```

  - ...and later in the program,

```
myArray = new int[500];     // legal!
```

  - This is legal because the array's *size* is not part of its *type,* but part of its *value*

Length of an array

- Arrays are objects
- Every array has an instance constant, `length`, that tells how large the array is
- Example:

```
for (int i = 0; i < scores.length; i++)
    System.out.println(scores[i]);
```

- Use of `length` is always preferred over using a constant such as `10`
- Arrays have a `length` *variable*, Strings have a `length()` *method*

Stepping through an array

- The `for` loop is ideal for visiting every value in an array
- The form is: `for (int i = 0; i <` ***myArray***`.length; i++) {…}`
- Example:

```
for (int i = 0; i < students.length; i++) {
    System.out.println(students[i].name);
}
```

- In general we like to use meaningful names for variables, but in this case, the name `i` is traditional, and better
  - `i` is instantly recognizable as the index of an enclosing `for` loop
  - Inner (nested) loops should use `j`, then `k` (then, if necessary, `m`, then `n`, but *not* `l`– do you see why?)
    - You should avoid deeply nested loops—three is deep enough!
- It's usually best to declare `i` right in the `for` statement itself
- `i++` means "add 1 to `i`"

Example of array use I

- Suppose you want to find the largest value in an array `scores` of 10 integers:

```
int largestScore = 0;
for (int i = 0; i < 10; i++) {
    if (scores[i] > largestScore) {
        largestScore = scores[i];
    }
}
```

- Would this code work if next time you had 12 scores? Or 8 scores?

- Do you see an error in the above program?

Example of array use II

- To find the largest value in an array scores of (possibly negative) integers:

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

Example of array use III

- Suppose you want to find the largest value in an array scores *and* the location in which you found it:

```
int largestScore = scores[0];
int index = 0;
for (int i = 1; i < scores.length; i++) {
    if (scores[i] > largestScore) {
        largestScore = scores[i];
        index = i;
    }
}
```

Array names

- The names of array variables should be camelCase
  - Use lowercase for the first word and capitalize only the first letter of each subsequent word that appears in a variable name
- Array names should (usually) be *plural* nouns
- Example array names:

```
scores
phoneNumbers
preferredCustomers
```

Magic numbers

- Use names instead of numbers in your code
  - Names help document the code; numbers don't
  - It may be hard to tell why a particular number is used--we call it a magic number
    - This is a pejorative term
  - You might change your name about the value of a "constant" (say, more than ten `scores`)
    - You can change the value of a name in *one* place
  - An array's `length` is always correct!

NullPointerException

- Suppose you *declare* a Person p but you don't *define* it
  - Until you define p, it has the special value null
  - null is a legal value for any kind of object
  - null can be assigned, tested, and printed
  - But if you try to use a field or method of null, such as p.name or p.birthday(), the error you get is a nullPointerException

Arrays of objects

- Suppose you declare and define an array of objects:
  - `Person[ ] people = new Person[20];`
- You have given a value to the *array* named `people`, but you haven't yet given values to each *element* in that array
- There is nothing wrong with this array, but
  - it has 20 *references* to Persons in it
  - all of these references are initially `null`
  - you have *not yet* defined 20 Persons
  - For example, `people[12].name` will give you a `nullPointerException`

Initializing arrays I

- Here's one way to initialize an array of objects

```
Person[ ] people = new Person[20];
for (int i = 0; i < people.length; i++) {
    people[i] = new Person("Dave");
}
```

- This approach has a slight drawback: all the array elements have similar values

Initializing arrays II

- There is a special syntax for giving initial values to the elements of arrays
  - This syntax can be used in place of `new` ***type*[*size*]**
  - It can *only* be used in an *array declaration*
  - The syntax is: { ***value***, ***value***, ..., ***value*** }
- Examples:

```
int[ ] primes = { 2, 3, 5, 7, 11, 13, 19 };
String[ ] languages = { "Java", "C", "C++" };
```

Array literals

- You can create an array literal with the following syntax:

  ***type*[ ] { *value1*, *value2*, …, *valueN* }**

- Examples:

```
myPrintArray(new int[] {2, 3, 5, 7, 11});

int[ ] foo;
foo = new int[]{42, 83};
```

Initializing arrays III

- To initialize an array of Person:

```
Person[ ] people =
    { new Person("Alice"),
      new Person("Bob"),
      new Person("Carla"),
      new Person("Don") };
```

- Notice that you do *not* say the size of the array
  - The computer is better at counting than you are!

Arrays of arrays

- The elements of an array can themselves be arrays
- Once again, there is a special syntax
- Declaration: `int[ ][ ] table;` (or `int table[ ][ ];`)
- Definition: `table = new int[10][15];`
- Combined: `int[ ][ ] table = new int[10][15];`
- The first index (10) is usually called the row index; the second index (15) is the column index
- An array like this is called a two-dimensional array

Example array of arrays

- `int[ ][ ] table = new int[3][2];` or,
- `int[ ][ ] table = { {1, 2}, {3, 6}, {7, 8} };`

	0	1
0	1	2
1	3	6
2	7	8

- For example, `table[1][1]` contains 6
- `table[2][1]` contains 8, and
- `table[1][2]` is "array out of bounds"
- To "zero out this table":

```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 2; j++)
        table[i][j] = 0;
```

- How could this code be improved?

Size of 2D arrays

	0	1
0	1	2
1	3	6
2	7	8

- `int[ ][ ] table = new int[3][2];`
- The length of this array is the number of *rows:* `table.length` is `3`
- Each row contains an array
- To get the number of *columns,* pick a row and ask for its length:

  `table[0].length` is `2`

  - Most of the time, you can assume all the rows are the same length
    - However: `table[2] = new int[50];`

The End

“All programmers are optimists. Perhaps this modern sorcery especially attracts those who believe in happy endings and fairy godmothers. Perhaps the hundreds of nitty frustrations drive away all but those who habitually focus on the end goal.

“Perhaps it is merely that computers are young, programmers are younger, and the young are always optimists. But however the selection process works, the result is indisputable: ‘This time it will surely run’ or ‘I just found the last bug’.”

--Fred Brooks