

Arrays

To keep track of 10 exam scores, we could declare 10 separate variables: `int score1, score2, score3, ... , score10`; But what if we had 100 exam scores? That would be a lot of variables! Most programming languages have a simple **data structure** for a collection of related data that makes this easier. In Python, this is called a list. In Java and many programming languages, this is called an **array**.

An **array** is a block of contiguous memory that stores a collection of data items (**elements**) of the same type under one name. Arrays are useful whenever you have many elements of data of the same type that you want to keep track of, but you don't need to name each one. Instead, you use the array name and a number (called an **index**) for the position of an item in the array. You can make arrays of ints, doubles, Strings, and even classes that you have written like Students.

Here's a fun [video](#) that introduces the concept of an array and gives an example.

When we declare a variable in Java, we specify its type and then the variable name. To make a variable into an array, we put square brackets after the data type. This tells the compiler to allocate multiple blocks of memory of that type.

```
// Declaration for a single int variable
int score;
// Declaration for an array of ints
int[] scores;
```

The declarations do not create the array. Arrays are **objects** in Java, so any variable that declares an array **holds a reference to an object**. If the array hasn't been created yet and you try to print the value of the variable, it will print **null** (meaning it doesn't reference any object yet).

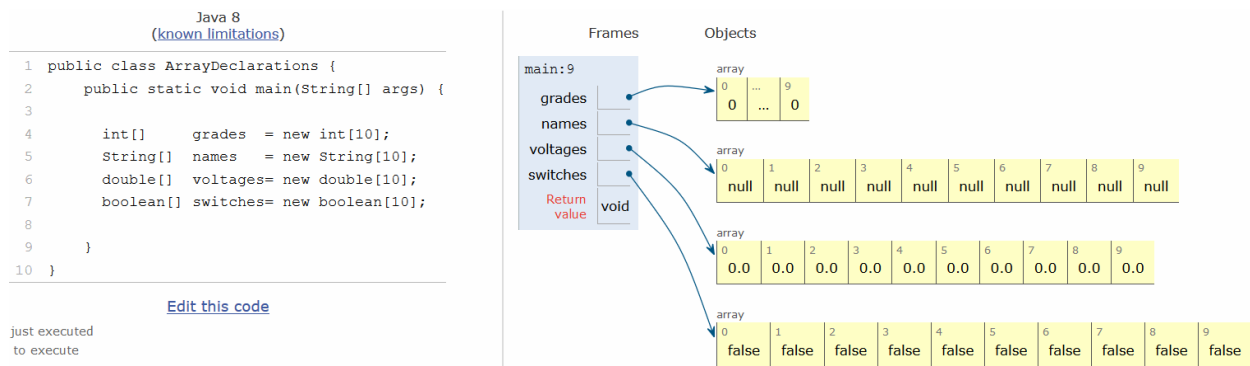
To actually create an array after declaring the variable, use the **new** keyword with the type and the size of the array (the number of elements it can hold). This will actually create the array in memory. You can do the declaration and the creation all in one step, see the String array names below. *The size of an array is set at the time of creation and cannot be changed after that.*

```
//declare an array variable
int[] highScores;
// create the array using new
highScores = new int[5];
// declare and create array in 1 step!
String[] names = new String[5];
```

Note

Array elements are initialized to default values like the following.

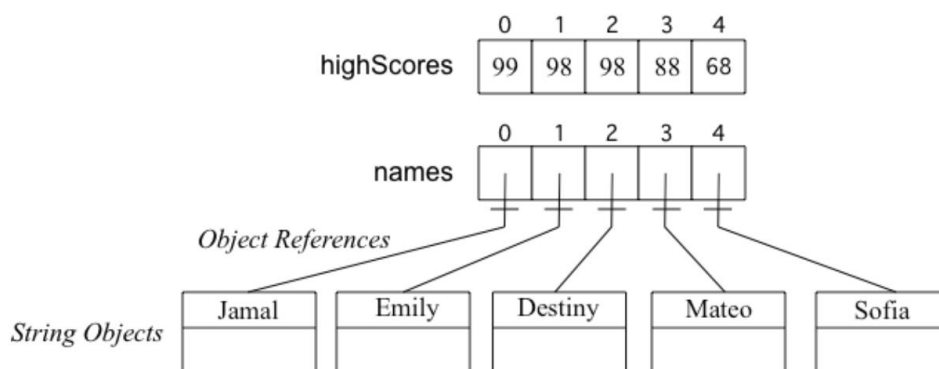
- 0 for elements of type **int**
- 0.0 for elements of type **double**
- false for elements of type **boolean**
- null for elements of type **String**



Notice that the array variables simply *point to* the values in memory. An array variable just contains a number, and that number is the address of the array in memory.

Another way to create an array is to use an **initializer list**. You can initialize (set) the values in the array to a list of values in curly brackets { } when you create it, like below. In this case you don't specify the size of the array, it will be determined from the number of values that you specify. You also do not have to use the **new** keyword. The compiler will insert this automatically.

```
int[ ]    highScores = {99, 98, 98, 88, 68};  
String[ ] names      = {"Jamal", "Emily", "Destiny", "Mateo", "Sofia"};
```



Arrays know their length (how many elements they can store). It is a public read-only instance variable so you can use **dot-notation** to access the instance variable (arrayName.length). **Dot-notation** is using variable name followed by a . and then the instance variable (property) name or a method name.

```
[jshell> int[ ] highScores = {99,98,98,88,68};  
highScores ==> int[5] { 99, 98, 98, 88, 68 }  
  
[jshell> System.out.println(highScores.length);  
5
```

Note: Note that length is an instance variable and not a method, unlike the String *length()* method, so you don't add parentheses after length. The length instance variable is declared as a public final int. *public* means you can access it and *final* means the value can't change.

Yes, this is confusing considering a String's length is gotten by calling a method.

```
[jshell> String name = "Frank N. Stein";  
name ==> "Frank N. Stein"  
  
[jshell> System.out.println(name.length());  
14
```

Array Element Access

To access the items in an array, we use an *indexed array variable* which is the array name and the index inside of square bracket []. Remember that an index is a number that indicates the position of an item in a list, starting at 0.

An indexed variable like *arrayname[index]* can be used anywhere a regular variable can be used, for example to assign a new value or to get a value from the array like below.

```
// assign a new value 99 to the first element in the array  
highScores[0] = 99;  
// print the first element of the array  
System.out.println( highScores[0] )
```

Note: The first value in an array is stored at *index 0* and the index of the last value is the length of the array minus one (since the first index is 0). Use *arrayname[index]* to access or modify array items.

ArrayIndexOutOfBoundsException

If you try to access an array index that is out of the range of $\{0 \rightarrow \text{array.length} - 1\}$ the program will throw an *ArrayIndexOutOfBoundsException* and crash. Take special care to keep array indices within range. To quote the Java documentation, an exception of this type is . . .

“Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.”

We do not try/catch these issues as they are always the result of bad code.

```
[jshell> int[] array = {1, 2, 3, 4, 5};  
array ==> int[5] { 1, 2, 3, 4, 5 }  
  
[jshell> int n = array[array.length];  
| Exception java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5  
| at (#6:1)  
  
[jshell> int n = array[array.length - 1];  
n ==> 5
```

Parallel Arrays

A group of parallel arrays is a form of implicit data structure that uses multiple arrays to represent a singular array of records. It keeps a separate, homogeneous data array for each field of the record, each having the same number of elements. Then, objects located at the same index in each array are implicitly the fields of a single record

In the following image there are 3 arrays to track prices, quantities and revenues for products. The index values across each of the arrays form a **record** or a collection of related items of information (as in a database) treated as a single unit

price[]		quantity[]		revenue[]
1.99	*	56	=	111.44
4.95	*	38	=	188.10
2.99	*	42	=	125.58
14.95	*	20	=	299.00
28.95	*	17	=	492.15

Index 0 across each of the parallel arrays creates a **record**

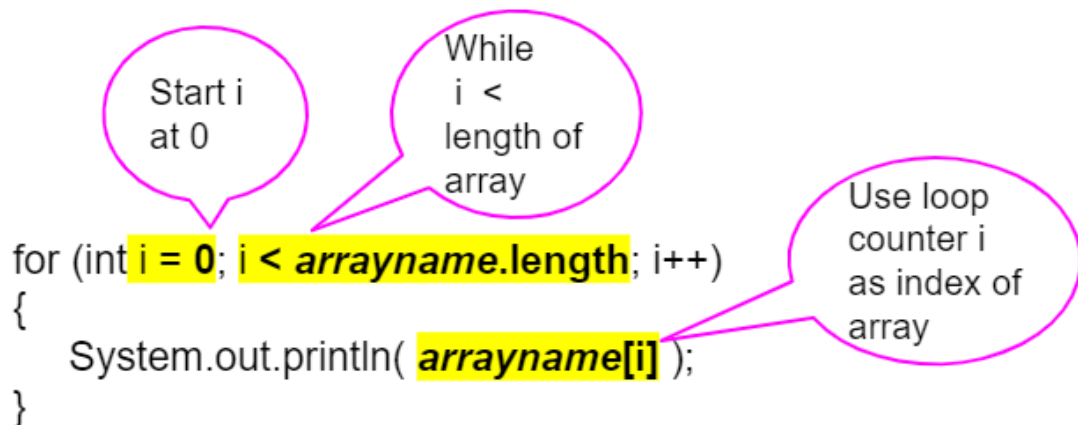
In this situation parallel arrays are necessary as the data types for each item are different. Prices and revenue are doubles and quantities are integers. Another way of dealing with this would be

to define the class **Product** and encapsulate these data and declare an array of Product type. We will come to that.

Check the program *ParallelArrays.java* for a full example

Array Traversal

We can use iteration with a for loop to visit each element of an array. This is called *traversing* the array. Just start the index at 0 and loop while the index is less than the length of the array. Note that the variable *i* (short for index) is often used in loops as the loop counter variable and is used here to access each element of an array with its index.



```
[jshell> int[] highScores = { 10, 9, 8, 11};  
highScores ==> int[4] { 10, 9, 8, 11 }  
  
jshell> for (int i = 0; i < highScores.length; i++){  
...>     System.out.println( highScores[i] );  
[ ...> }  
10  
9  
8  
11
```

Note: Using a variable as the index is a powerful data abstraction feature because it allows us to use loops with arrays where the loop counter variable is the index of the array! This allows our code to generalize to work for the whole array. There is some address manipulation magic that happens behind the scenes.

When processing all array elements, be careful to start at the first index which is 0 and end at the last index. Usually, loops are written so that the index starts at 0 and continues while the index is less than *arrayName.length* since (*arrayName.length* - 1) is the index for the last element in the

array. **Make sure you do not use `<=` instead of `<`** If the index is less than 0 or greater than (arrayName.length - 1), an *ArrayIndexOutOfBoundsException* will be thrown. **Off by one** error, where you go off the array by 1 element, are easy to make when traversing an array which result in an *ArrayIndexOutOfBoundsException* being thrown.

Enhanced for loop

There is a special kind of loop that can be used with arrays called an *enhanced for loop* or a *for each loop*. This loop structure achieves the same level of abstraction as Python's for loop. This loop is much easier to write because it does not involve an index variable or the use of the []. It just sets up a variable that is set to each value in the array successively.

To set up a for-each loop, use *for (type variable : arrayname)* where the type is the type for elements in the array, and read it as "for each variable value in arrayname".

```
for (type item : array) {  
    // statements using item;  
}
```

```
[jshell> for (String name : names)  
[    ...>     System.out.println("Name is: " + name);  
Name is: Jamal  
Name is: Emily  
Name is: Destiny  
Name is: Mateo
```

Drawbacks of for each loops

Use the enhanced for each loop with arrays whenever you can, because it cuts down on errors. You can use it whenever you need to loop through all the elements of an array and don't need to know their index and don't need to change their values. It starts with the first item in the array (the one at index 0) and continues through in order to the last item in the array. This type of loop can only be used with arrays and some other collections of items like ArrayLists which we will be covering.

What if we had a loop that incremented all the elements in the array. Would that work with an enhanced for-each loop? Unfortunately not! Because only the variable in the loop changes, not the real array values. We would need an indexed loop to modify array elements.

Enhanced for each loops cannot be used in all situations. Only use for-each loops when you want to loop through **all** the values in an array without changing their values.

- Do not use for each loops if you need the index
- Do not use for each loops if you need to access multiple array elements simultaneously
- Do not use for each loops if you need to change the values in the array.
- Do not use for each loops if you want to loop through only part of an array or in a different order.

Arrays of Objects

When we require a single object to store in our program, we do it with a variable of type Object. But when we deal with numerous objects, then it is preferred to use an Array of Objects. Objects in Java are anything with an associated class definition. It could be an array of Strings, an array of Scanners or an array of a user defined class (more on this later).

Let's dig a little deeper into the concept of an array of objects using the String class. As we have seen, you can create an array of objects just like you can an array of primitives. The code below creates an array of 10 String objects.

```
// create an array of 10 String objects  
String[] names = new String[10];
```

It is important to understand that upon creation this array hold 10 ***null pointers***. All we have done is allocate memory for 10 references to Strings. The strings themselves do not exist yet . . . just the container that will eventually hold them. You can see this by iterating through the array and printing each String object.

```
StringArray.java ×
StringArray.java > ...
1 public class StringArray{
    Run | Debug
2     → public static void main(String[] args){
3
4     → → // create an array of 10 String objects
5     → → String[] names = new String[10];
6     → → // this array initially contains 20 null pointers
7     → → for (String name : names)
8     → → → System.out.println(name); // will print null
9     → → }
10 }

OUTPUT  TERMINAL  DEBUG CONSOLE  PROBLEMS

> javac StringArray.java
> java StringArray
null
null
null
null
null
null
null
null
null
null
null
```

Let's fill the array with some names from a text file and see how to interact with these objects once they are inside an array. I have a text file of random names.

Check out the method *readNames* below (review the *Static Method* document for a full description of methods). This method accepts the array and fills it from the external file. It is **void** because arrays are passed *by reference* and are *mutable*. The method can simply fill the array that was provided.

```
StringArray.java  names.txt ×
names.txt
1 Franchot Crayk
2 Tova Sheirlaw
3 Owen Hollingshead
4 Dru Issac
5 Andres Staveley
6 Leigha Haestier
7 Enid Hawthorne
8 Franky Maclean
9 Bree Kwietak
10 Gun Gutowski
```



```

public static void readNames(String[] n){
    try{
        File text = new File("names.txt"); // create a File instance
        Scanner fileReader = new Scanner(text); // create a Scanner instance

        int index = 0; // variable to track the array index
        while(fileReader.hasNext()){ // as long as there is more to read
            n[index] = fileReader.nextLine(); // read the entire line
            ++index; // increment the index
        }
        fileReader.close(); // close the scanner
    }
    catch(FileNotFoundException fnfe){
        System.out.println("There was a problem with the file");
    }
}

```

We can now call the method, pass the array and print the results

```

// create an array of 10 String objects
String[] names = new String[10];
// call readNames method and pass the array
readNames(names);
// now try to print again
for(String name : names)
    System.out.println(name);

```

This gives the following output

```

> javac StringArray.java
> java StringArray
Franchot Crayk
Tova Sheirlaw
Owen Hollingshead
Dru Issac
Andres Staveley
Leigha Haestier
Enid Hawthorne
Franky Maclean
Bree Kwietak
Gun Gutowski

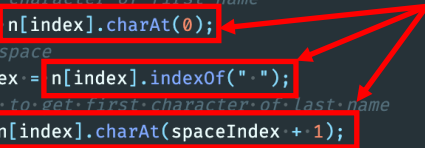
```

Let's now write some code to print the initials (first letter of first and last name). This will require

1. Get the first character of the first name
2. Find the space
3. Use the location of the space to get the first character of the last name

The important concept here is that you need to invoke method calls on the objects that are contained in the array. I will illustrate this with an *explicit for loop* so you can see the underlying mechanics.

```
public static void printInitials(String[] n){  
    // iterate through the array. Use an explicit for loop to show the mechanics  
    for (int index = 0; index < n.length; ++index){  
        // get first character of first name  
        char first = n[index].charAt(0);  
        // find the space  
        int spaceIndex = n[index].indexOf(" ");  
        // use space to get first character of last name  
        char last = n[index].charAt(spaceIndex + 1);  
        // print the results  
        System.out.println(n[index] + "'s initials are: " + String.valueOf(first) + String.valueOf(last));  
    }  
}
```



Method calls on objects inside of an array

This generates the following output

```
5 public class StringArray{  
    Run | Debug  
6 → public static void main(String[] args){  
7 → → // create an array of 10 String objects  
8 → → String[] names = new String[10];  
9 → → // call readNames method and pass the array  
10 → → readNames(names);  
11 → → // now try to print again  
12 → → for (String name : names)  
13 → → → System.out.println(name);  
14  
15 → → // call printInitials  
16 → → printInitials(names);  
}
```

```
> javac StringArray.java  
> java StringArray  
Franchot Crayk  
Tova Sheirlaw  
Owen Hollingshead  
Dru Issac  
Andres Staveley  
Leigha Haestier  
Enid Hawthorne  
Franky Maclean  
Bree Kwietak  
Gun Gutowski  
Franchot Crayk's initials are: FC  
Tova Sheirlaw's initials are: TS  
Owen Hollingshead's initials are: OH  
Dru Issac's initials are: DI  
Andres Staveley's initials are: AS  
Leigha Haestier's initials are: LH  
Enid Hawthorne's initials are: EH  
Franky Maclean's initials are: FM  
Bree Kwietak's initials are: BK  
Gun Gutowski's initials are: GG
```

Null Pointer Exception

If the objects inside of an array have not been properly created and you attempt to call a method via an *array reference* you will get a *NullPointerException* as shown in this JShell snippet.

```
[jshell> String[] names = new String[10];
names ==> String[10] { null, null, null, null, null, null, null, null, null, null }

[jshell> char initial = names[0].charAt(0);
|   Exception java.lang.NullPointerException
|   at (#3:1)
```

This can be a real pesky bug to track down. Pay close attention to this!!

Two Dimensional Arrays (Arrays of Arrays)

A two-dimensional array is an array where each element is also an array. It is helpful to view this structure as two-dimensional table that consists of an intersection of rows and columns.

This code `int[][] table = new int[3][4]`; creates the following table:

	Column 1	Column 2	Column 3	Column 4
Row 1	<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[0][3]</code>
Row 2	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[1][3]</code>
Row 3	<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[2][3]</code>

Make certain you understand the dual indexing necessary to identify a cell.

Initializing an 2D Array

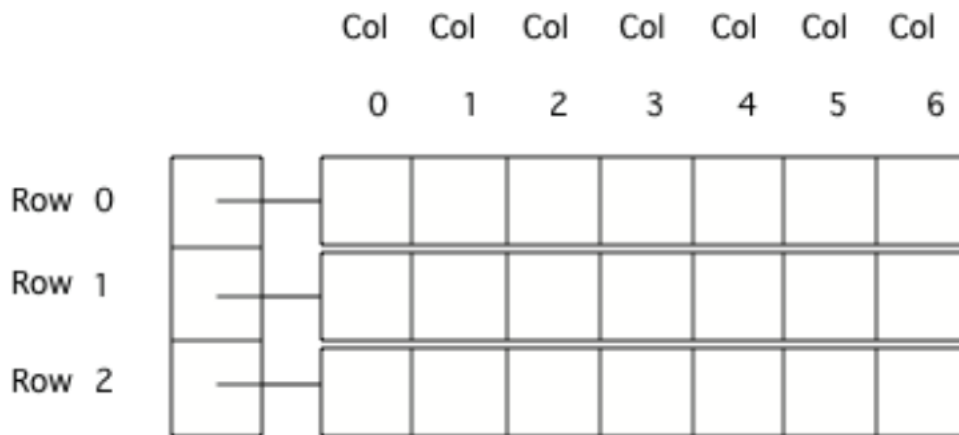
2D arrays can be initialized just like regular arrays

```
int[ ][ ] a = {
    {1, 2, 3},
    {4, 5, 6, 9},
    {7}
};
```

The statement above will create the following situation

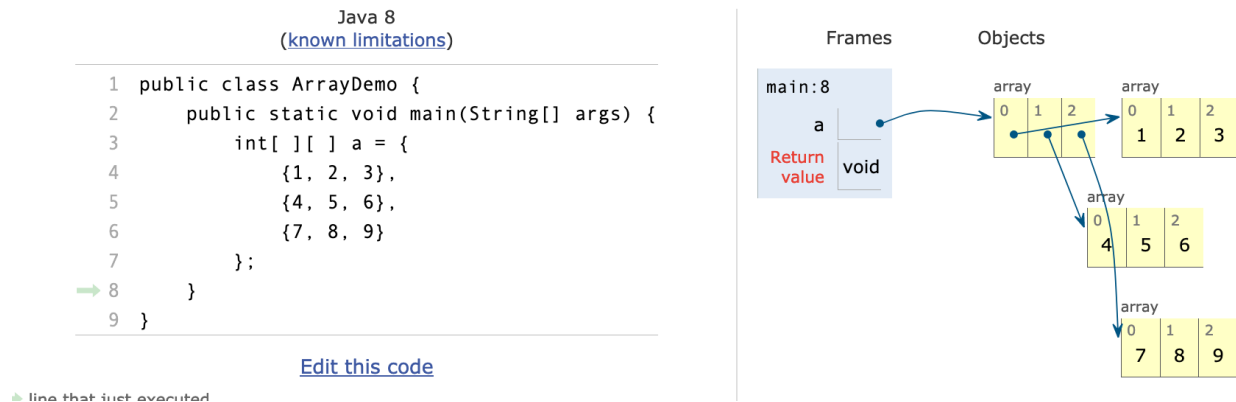
	Column 1	Column 2	Column 3	Column 4
Row 1	<div>1</div> <div>a[0][0]</div>	<div>2</div> <div>a[0][1]</div>	<div>3</div> <div>a[0][2]</div>	
Row 2	<div>4</div> <div>a[1][0]</div>	<div>5</div> <div>a[1][1]</div>	<div>6</div> <div>a[1][2]</div>	<div>9</div> <div>a[1][3]</div>
Row 3	<div>7</div> <div>a[2][0]</div>			

Java actually stores two-dimensional arrays as arrays of arrays. Each element of the outer array has a reference to each inner array. The picture below shows a 2D array that has 3 rows and 7 columns. Notice that the array indices start at 0 and end at the length - 1.



Python (Java) Tutor makes this very clear

Notice that there are actually four arrays here. The array variable **a** holds the references to the inner arrays of ints; it is not actually stored in memory as a table. The inner arrays do not have official names. Their reference is stored as an element in the outer array.



Declaring 2D Arrays

To declare a 2D array, specify the type of elements that will be stored in the array, then ([][]) to show that it is a 2D array of that type, then at least one space, and then a name for the array. Note that the declarations below just name the variable and say what type of array it will reference.

The declarations do not create the array. Arrays are objects in Java, so any variable that declares an array holds a reference to an object. If the array hasn't been created yet and you try to print the value of the variable, it will print null (meaning it doesn't reference any object yet).

```
[jshell> int[][] ticketInfo;
ticketInfo ==> null

[jshell> String[][] seatingChart;
seatingChart ==> null
```

To create an array use the new keyword, followed by a space, then the type, and then the number of rows in square brackets followed by the number of columns in square brackets, like this

new int[numRows][numCols];

The code below creates a 2D array with 2 rows and 3 columns named ticketInfo and a 2D array with 3 rows and 2 columns named seatingChart. The number of elements in a 2D array is the number of rows times the number of columns.

```
[jshell> ticketInfo = new int[2][3];
ticketInfo ==> int[2][] { int[3] { 0, 0, 0 }, int[3] { 0, 0, 0 } }

[jshell> seatingChart = new String[3][2];
seatingChart ==> String[3][] { String[2] { null, null }, String[2] ... String[2] { null, null } }
```

Notice that because Strings are objects in Java that the 2D array of strings is initialized to null. No actual strings have been created yet.

Setting Values in a 2D Array

When arrays are created their contents are automatically initialized to 0 for numeric types, null for object references, and false for type boolean. To explicitly put a value in an array, you can use assignment statements with the name of the array followed by the row index in brackets followed by the column index in brackets and then an = followed by a value.

```
[jshell> int[][] ticketInfo = new int[2][3];
ticketInfo ==> int[2][] { int[3] { 0, 0, 0 }, int[3] { 0, 0, 0 } }

[jshell> ticketInfo[0][0] = 15;
$28 ==> 15

[jshell> System.out.println(ticketInfo[0][0]);
15
```

2D Array Initializer Lists

You can also initialize (set) the values for the array when you create it. In this case you don't need to specify the size of the array, it will be determined from the values you give. You also do not have to use the **new** keyword. The code below creates an array called *ticketInfo* with 2 rows and 3 columns. It also creates an array called *seatingInfo* with 3 rows and 2 columns.

```
[jshell> int[][] ticketInfo = { {25,20,25}, {25,20,25} };
ticketInfo ==> int[2][] { int[3] { 25, 20, 25 }, int[3] { 25, 20, 25 } }

[jshell> String[][] seatingInfo = { {"Jamal", "Maria"},
...>                                {"Jake", "Suzy"},
...>                                {"Emma", "Luke"}
[ ...>                                };
seatingInfo ==> String[3][] { String[2] { "Jamal", "Maria" }, Str ... ng[2] { "Emma", "Luke" } }
```

Accessing 2D array elements

To get the value in a 2D array give the name of the array followed by the row and column indices in square brackets. The code below will get the value at row index 1 (second row) and column index 2 (third column) from *ticketInfo*. It will also get the value at row index 1 and column index 1 from *seatingChart*.

```
jshell> String[][] seatingInfo = { {"Jamal", "Maria"},
...>                               {"Jake", "Suzy"},
...>                               {"Emma", "Luke"}
[...>                               };
seatingInfo ==> String[3][] { String[2] { "Jamal", "Maria" }, Str ... ng[2] { "Emma", "Luke" } }

jshell> int ticket = ticketInfo[1][2];
ticket ==> 25

jshell> String passenger = seatingInfo[1][1];
passenger ==> "Suzy"
```

Getting the number of rows and columns

Arrays know their length (how many elements they can store). The length is a public read-only field so you can use *dot-notation* to access the field (***arrayName.length***). The length of the outer array is the number of rows and the length of one of the inner arrays is the number of columns (***arrayName[0].length***)

```
jshell> ticketInfo.length; // returns the number of rows
$39 ==> 2

jshell> ticketInfo[0].length // returns the number of columns for the first row
$40 ==> 3
```

Traversing a 2D Array

Since you can find out the number of rows and columns in a 2D array you can use a ***nested for loop*** (one loop inside of another loop) to loop/traverse through all of the elements of a 2D array.

Here is a program that illustrates this concept. You can find the source code file with the reading material and here is a link to a ***PythonTutor (Java)*** session that you can step through. This is strongly recommended.

<https://tinyurl.com/yyc8gnqe>

Code follows on the next page

```

1  public class TwoDArrayTraversal {
    Run | Debug
2  → public static void main(String[] args) {
3  → → // declare and initialize a 2D array of ints
4  → → int[][] array = {
5  → → → {1,2,3,4},
6  → → → {5,6,7,8}
7  → → → };
8
9  → → // outer loop to traverse the number of "rows"
10 → → for (int row = 0; row < array.length; row++) {
11 → → → // print a row header
12 → → → System.out.print("Row " + (row + 1) + ": ");
13 → → → // inner loop to traverse the number of "columns"
14 → → → for (int col = 0; col < array[0].length; col++)
15 → → → → // print a single item without line break
16 → → → → System.out.print(array[row][col] + " ");
17 → → → → // print a line break
18 → → → → System.out.println();
19 → → } // end outer loop
20 → } // end main
21 } // end class

```

Some key things to notice about this code are:

- The number of rows is *array.length*
- The number of columns is *array[0].length*
- The number of times this loop executes is the number of rows times the number of columns.
- **Line 12** will print once for each row. Notice that it exists *before* the inner loop, so it will execute *array.length* times
- The inner loop uses *System.out.print* (without a line break) so that each value in “column” will appear on the same line.
- **Line 18** prints a line break to signal the end of a row.

Here is an additional example that computes the average of a 2D Array

PythonTutor session: <https://tinyurl.com/y3brbбен>

```
1 public class TwoDArrayAverage{
2     /*
3     → Method accepts a 2D array and returns a double
4     */
5     public static double getAverage(int[][] a){
6     → double total = 0;
7     → //outer loop to traverse rows
8     → for(int row = 0; row < a.length; row++){
9     → → //inner loop to traverse columns
10    → → for(int col = 0; col < a[0].length; col++){
11    → → → //access array element and accumulate a total
12    → → → total += a[row][col];
13    → → }
14    → //do math and return the result
15    → return total / (a.length * a[0].length);
16    }
17
18    Run | Debug
19    public static void main(String[] args){
20    → //declare and initialize array
21    → int[][] matrix = {{1,2,3},{4,5,6}};
22    → //call method and store result locally
23    → double avg = getAverage(matrix);
24    → //print the result
25    → System.out.println("Average is: " + avg);
26    }
27 }
```

Some key things to notice about this code are:

- **total** is declared to be a double so that the result will be a double. If total was declared to be an int then the result would be an integer and the values after the decimal point would be thrown away.
- A reference to the array is passed as an argument to the method. This is important to understand. The array itself is not passed . . . just it's address in memory.
- The number of rows is **a.length**
- The number of columns is **a[0].length**
- The number of times this loop executes is the number of rows times the number of columns.

Black Bear Sightings: You are helping the forest service in Alaska to track the number of black bear sightings. You have a collection data from seven different regions. This data spans 12 months and you are given the totals for each month, from each region. You need to compute the following

1. Average black bear sightings per month
2. Average black bear sightings per region

The data has been exported to a comma separated file of the following structure. Each line in the file represents the data from a region. Each region has 12 data points, one for each month.

```
black_bear_sightings.txt
41,3,4,47,70,69,56,88,40,34,5,76
1,92,35,30,88,9,49,40,100,32,59,82
86,78,48,77,93,60,24,9,85,81,61,77
47,38,3,43,35,42,39,13,26,22,4,85
55,21,21,41,2,52,16,93,43,16,74,28
12,91,93,31,10,83,20,15,78,21,35,84
67,23,12,14,66,78,59,7,48,38,66,72
```

Design Choices

1. When the program launches read the data from the file into a 2-dimensional array and then close the file. All calculations will happen on the array.
2. Regions will be numbered as *Region 1, Region 2, ... Region N*
3. Months will be labeled by their name.
4. Full code is available in the git repo

Operations

We now have a two-dimensional table (matrix) and we need to process this table by column and by row. To get the average per month we need to sum each column. To get the average per region we need to sum each row.

	0	1	2	3	4	5	6	7	8	9	10	11
0	41	3	4	47	70	69	56	88	40	34	5	76
1	1	92	35	30	88	9	49	40	100	32	59	82
2	86	78	48	77	93	60	24	9	85	81	61	77
3	47	38	3	43	35	42	39	13	26	22	4	85
4	55	21	21	41	2	52	16	93	43	16	74	28
5	12	91	93	31	10	83	20	15	78	21	35	84
6	67	23	12	14	66	78	59	7	48	38	66	72

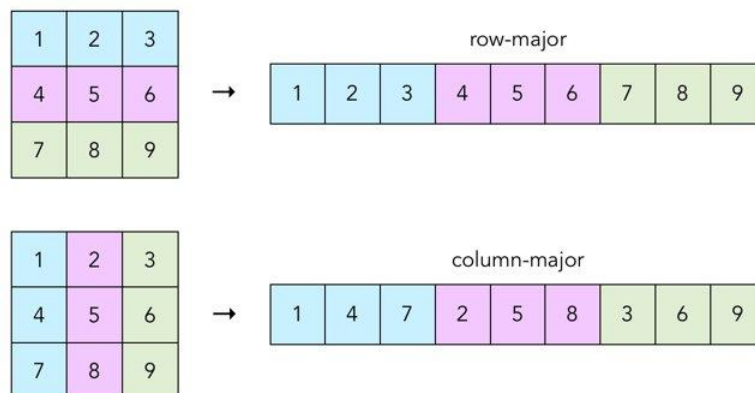
Average for Region One is: 44.41

Average for January is: 44.14

Row Major and Column Major Ordering of Arrays

In computing, row-major order and column-major order are methods for storing multidimensional arrays in linear storage such as random-access memory.

The difference between the orders lies in which elements of an array are contiguous in memory. In row-major order, the consecutive elements of a row reside next to each other, whereas the same holds true for consecutive elements of a column in column-major order. While the terms allude to the rows and columns of a two-dimensional array, i.e. a matrix, the orders can be generalized to arrays of any dimension by noting that the terms row-major and column-major are equivalent to lexicographic and colexicographic orders, respectively.



Assume the following data definitions and that the sightings array is populated. You can access the full example in the module material

```
final int NUM_REGIONS    = 7;  
final int NUM_MONTHS     = 12;  
String fileName          = "";  
  
// 2D array to model the table  
int[][] sightings = new int[NUM_REGIONS][NUM_MONTHS];
```

The sightings array has 7 rows and 12 columns

Pay attention to the the configuration of the outer and inner loops here.

Column Major Order: Average sightings per month

```
// compute average per month
for(int i = 0; i < NUM_MONTHS; ++i) {
    double sum = 0.0;
    for(int j = 0; j < NUM_REGIONS; ++j)
        sum += sightings[j][i];
    System.out.printf("Average for %s:¥t %.2f ¥n", months[i], sum / NUM_REGIONS);
}
```

Row Major Order: Average sightings per region

```
// compute average per region
for(int i = 0; i < NUM_REGIONS; ++i) {
    double sum = 0;
    for(int j = 0; j < NUM_MONTHS; ++j)
        sum += sightings[i][j];
    System.out.printf("Average for Region %d:¥t %.2f ¥n", i+1, sum / NUM_MONTHS);
}
```