AP CSA

zhang si 张思

zhangsi@rdfz.cn

ICC 609

6- Array &ArrayList & 2D array

- Array
  - declare & initialize
  - access & modify
  - tranverse-enhanced for loop
- ArrayList
  - declare & initialize
  - methods
  - tranverse-enhanced for loop
  - develop algorithm with Array/Arraylist
- 2D array
  - declare & initialize
  - access & modify
  - tranverse-row-major & column major

Array Vocabulary

- An **array** is a data structure used to implement a collection (list) of primitive or object reference data.
- An **element** is a single value in the array.
- The **index** of an element is the position of the element in the array.
  - In Java, the first element of an array is at index **0**.
- The **length** of an array is the number of elements in the array.
  - `length` is a `public final` data member of an array
    - Since `length` is `public`, we can access it in any class!
    - Since `length` is `final`, we cannot change an array's `length` after it has been created
  - In Java, the last element of an array named `list` is at index `list.length - 1`

Array

- Arrays are objects that help us organize large amounts of information
- The values held in an array are called ***array elements***
- An array stores multiple values of the same type (the ***element type***)
- The element type can be a **primitive type** or an **object reference**
- Therefore, we can create an array of integers, or an array of characters, or an array of `String` objects, etc.
- In Java, the array itself is an **object**
- Therefore the name of the array is a object reference variable, and the array itself must be instantiated

Arrays

- An ***array*** is an ordered list of values, where the elements in the list are of the same type; for example, a class list of 25 test scores, a membership list of 100 names, or a store inventory of 500 items.

The entire array has a single name

Each value has a numeric *index*

value of scores[9]

scores

0	1	2	3	4	5	6	7	8	9
79	87	94	82	67	98	87	81	74	91

This array holds 10 values that are indexed from 0 to 9

An array of size N is indexed from zero to N-1

If a negative subscript is used, or a subscript k where k ≥ N, an ArrayIndexOutOfBoundsException is thrown

Declaring Arrays

- The `scores`

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

resevered word

type **reference variable** **size**

- using the `new` operator, the reference variable `scores` is set to a new array object that can hold 10 integers, **and the values of each slot is initialized to default values**

- Once an array is created, it has a fixed size,Each array object has a final public instance variable (ie, a constant) called length that stores the size of the array.It is referenced using the array name:

```
scores.length
```

- The size of an array remains **fixed** once it has been created. As with String objects, however, an array reference may be reassigned to a new array of a different size.

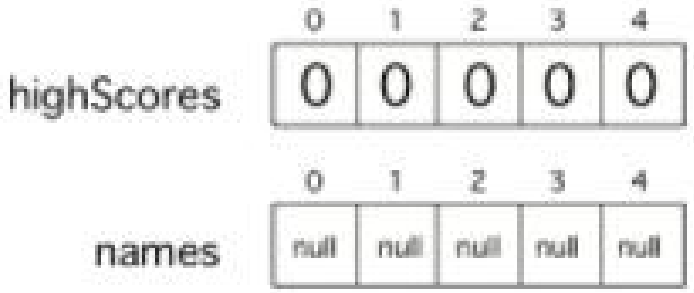
Declaring Arrays——default values

- when you creat arrays with `new` reserved word:

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`



Initializer Lists

- instead of writing

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

scores[0]=79;

scores[1]=87;

scores[2]=94;

scores[3]=82;

scores[4]=67;

scores[5]=98;

scores[6]=87;

scores[7]=81;

scores[8]=74;

scores[9]=91;
```

•An *initializer list* can be used to **instantiate and initialize** an array in one step

•The values are `delimited` by braces and separated by commas

```
int[]scores={79,87,94,82,67,98,87,81,74,91 };
```

•Note that when an initializer list is used:

- •the `new` operator is not used
- •no size value is specified

•The size of the array is determined by the number of items in the initializer list

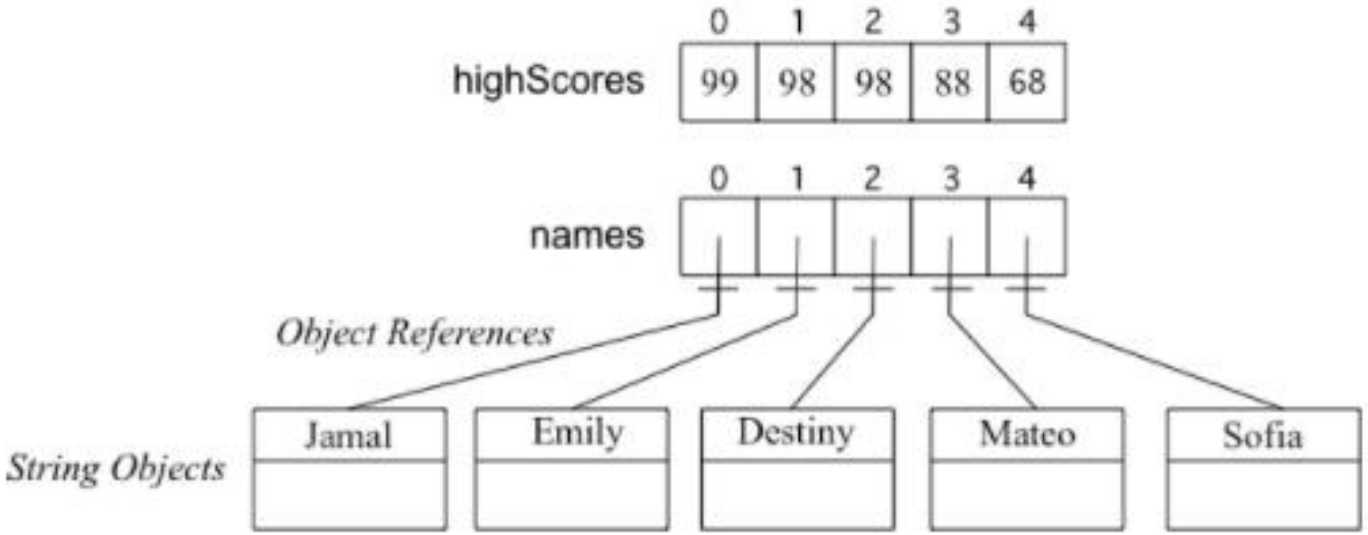
•An initializer list can only be used only in the array declaration

Arrays of Objects

- **The elements of an array can be object references**

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

When you create an array of a **primitive type** (like `int`) with initial values specified, space is allocated for the specified number of items of that type and the values in the array are set to the specified values. When you create an array of an **object type** (like `String`) with initial values, space is set aside for that number of object references. The objects are created and the object references set so that the objects can be found.



Arrays——access and modify

- A particular value in an array is referenced using the array name followed by the index in brackets

- For example, the expression

```
scores[2]
```

refers to the value 94 (the 3rd value in the array)

- That expression represents a place to store a single integer and can be used wherever an integer variable can be used

- Also， you can modify the value you access like this:

  - `scores[2]=90`

Index Variables

- we can use a variable for the index of an array.
- even do math with that index and have an arithmetic expression inside the [], like below.

```
// highScores array declaration
int[] highScores = { 10, 9, 8, 8};
// use a variable for the index
int index = 3;
// modify array value at index
highScores[index] = 11;
// print array value at index
System.out.println(  highScores[index] );
System.out.println(  highScores[index - 1] );
```

https://pythontutor.com/visualize.html#mode=display

- we can also use `Math.random()` method to generate index randomly (pay attention to the boundaries)
  - https://csawesome.runestone.academy/runestone/books/published/csawesome/Unit6-Arrays/topic-6-1-array-basics.html
  - Activity: 6.1.5.5 ActiveCode (imageArray)

For Loop to Traverse Arrays

- 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.

Start i at 0

While i < length of array

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

Use loop counter i as index of array

Attention!
if you want to print every elements of the array ,you need tranverse it!

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.

Enhanced for Loop for Arrays

```java
public class ForEachDemo
{
   public static void main(String[] args)
   {
     int[] highScores = { 10, 9, 8, 8};
     String[] names = {"Jamal", "Emily", "Destiny", "Mateo"};
     // for each loop with an int array
     for (int value : highScores)
     {
         System.out.println( value );
     }
     // for each loop with a String array
     for (String value : names)
     {
         System.out.println(value); // this time it's a name!
     }
   }
 }
```

results:

```
10
9
8
8
Jamal
Emily
Destiny
Mateo
```

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-each Loop Limitations

- Use the enhanced for each loop with arrays whenever you can, because it cuts down on errors.
- This type of loop can only be used with arrays and some other collections of items like ArrayLists which we will see in the next unit.

Note

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 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.

Difference between Array and ArrayList

Array	ArrayList
Fixed length	Resizable length
Fundamental Java feature	Part of a Framework
An Object with no methods	A Class with many methods
Not as flexible	Is designed to be flexible
Can store primitive data	Not designed to store primitives
	Is slightly slower than Arrays
	Can only be used with an import statement

Declaring and Creating ArrayLists

- To declare a ArrayList use ArrayList<Type> name Change the Type to be whatever type of objects you want to store in the ArrayList, for example String as shown in the code below. You don’t have to specify the **generic type** <Type>, since it will default to Object, but it is good practice to specify it to restrict what to allow in your ArrayList.
- Using a type ArrayList<Type> is preferred over just using ArrayList because it allows the compiler to find errors that would otherwise be missed until run-time.
- EG:

```
// ArrayList<Type> name = new ArrayList<Type>();

// An ArrayList of Strings:

ArrayList<String> shoppingList = new ArrayList<String>();
```

Primitive Values Disguised as `Wrapper` Class Objects

`ArrayList` objects are designed to only store references to objects, not primitive values. A workaround is to use `Wrapper` classes, which store primitive values as objects.

Primitive Data Types	Wrapper Class Data Types
boolean	Boolean
char	Character
double	Double
int	Integer

Note

ArrayLists can only hold objects like String and the wrapper classes Integer and Double. They cannot hold primitive types like int, double, etc.

ArrayList Methods you need to know

- int size()
- boolean add(E obj)
- void add(int index, E obj)
- E get(int index)
- E set(int index, E obj)
- E remove(int index)

ArrayList Methods

Size of the `ArrayList`

`int size()` : Returns the number of elements in the list

Consider the following code:

```
ArrayList<Integer> a1 = new ArrayList<Integer>();
```

The `ArrayList a1` has been instantiated with no entries.

```
System.out.println(a1.size());
```

result is : 0

```
ArrayList<Double> a2 = new ArrayList<Double>(15);
```

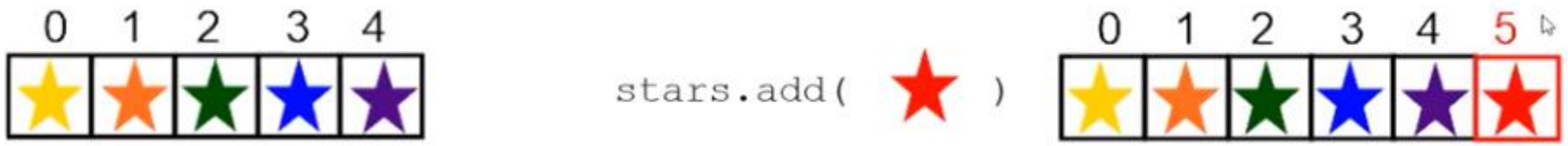
```
System.out.println(a2.size());
```

result is : 15

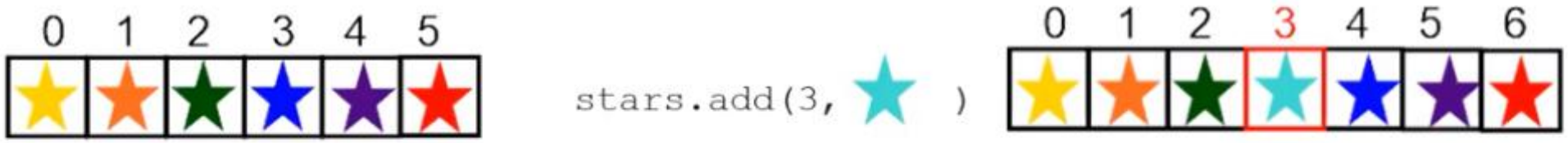
ArrayList Methods

Adding Items to an `ArrayList`

`boolean add(E obj)` : Appends `obj` to end of list; returns `true`.



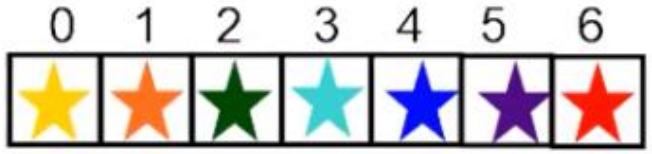
`void add(int index, E obj)` : Inserts `obj` at position `index` (0 ≤ `index` ≤ `size`), moving elements at position `index` and higher to the right (adds 1 to their indices) and adds 1 to list size.



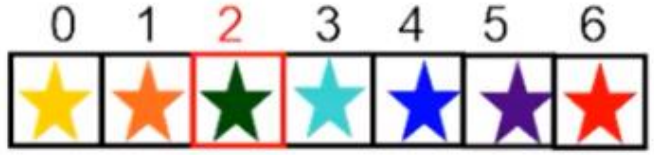
ArrayList Methods

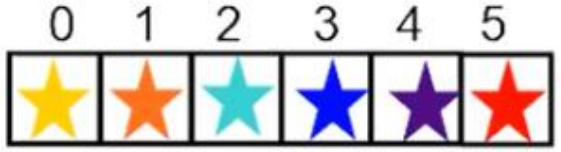
Deleting Items from an `ArrayList`

`E remove(int index)` : Removes element from position `index`, moves elements at position `index + 1` and higher to the left (subtracts 1 from their indices) and subtracts 1 from size; returns the element formerly at position `index`.

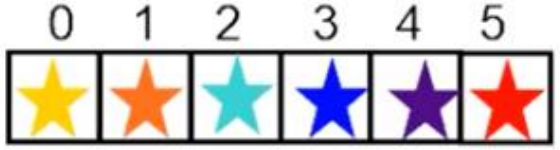



```
Star s1 = stars.remove( 2 );
```

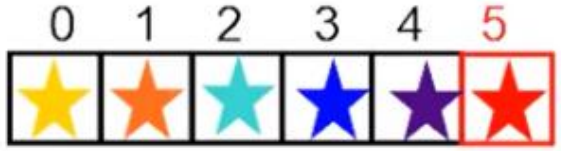


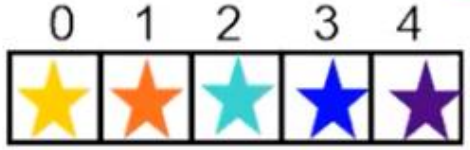



```
System.out.print(s1)
```




```
stars.remove(stars.size() - 1)
```

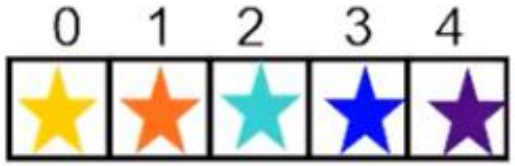




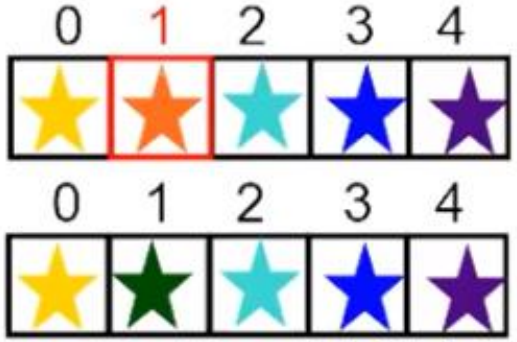
ArrayList Methods

Updating Items in an `ArrayList`

`E set(int index, E obj)` : Replaces the element at position `index` with `obj`; returns the element formerly at position `index`.




```
Star s1 = stars.set( 1, ★ );
```



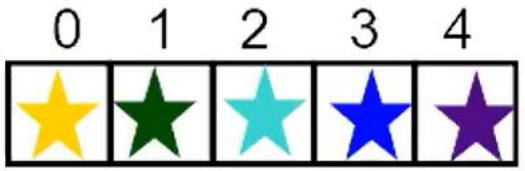

```
System.out.print(s1) ★
```

ArrayList Methods

Accessing Items in an `ArrayList`

`E get(int index)` : Returns the element at position `index` in the list.

`ArrayList<Star> stars` →




```
Star s1 = stars.get(1);
```
s1 ->

```
Star s2 = stars.get(0);
```
s2 ->

```
Star s2 = stars.get(stars.size()-1 );
```

- Enhanced for loop still works (same as array). You cannot add or remove items during enhanced for loop. (It's blocked.)
- When adding or **deleting elements** and running a loop, be careful not to skip elements or count them twice.

Possible `loop` Conditions to Traverse an `ArrayList`

	Initialization	Boolean Condition	Update
a)	`int i = 0;`	`i < arr.size();`	`i++;`
b)	`int i = 0;`	`i <= arr.size()-1;`	`i++;`
c)	`int i = arr.size()-1;`	`i >= 0;`	`i--;`
d)	`int i = arr.size()-1;`	`i > -1;`	`i--;`

Traversing: Accessing All the Elements

Suppose we have an **`ArrayList`** of Strings named **`roster`**, and we want to know the total number of characters in all of the Strings. In order to do this, we will need to visit **each** entry of the **`roster`** **`ArrayList`**.

```
int sum = 0;
for (int i=0; i <= roster.size()-1; i++)
{
        sum = sum + roster.get(i).length();
}
System.out.println( sum );
```

Retrieving the entry of **`roster`** at the ith index

Retrieving the number of characters at the ith index of **`roster`**

Enhanced `For` Loop Example

Suppose `roster` is an `ArrayList` of Strings of names of students and we want to determine the total number of characters needed to write out the roster of student names.

```
int sum = 0;
for (String name : roster )
{
    sum = sum + name.length();
}
```

Warning

During the iterations of an enhanced **`for`** loop, **`ArrayList`** elements **cannot** be modified, removed from, or added to the **`ArrayList`**.

Common Mistakes

- Forgetting to include `import java.util.ArrayList`
- Declaring and/or instantiating a `ArrayList` with a primitive data type

```
ArrayList<int> myList = new ArrayList<int>();
```

- Forgetting to include the () at the end of the **`ArrayList`** constructor call

```
ArrayList<Integer> myList = new ArrayList<Integer> ;
```

- Not specifying the element type that the **`ArrayList`** references

```
ArrayList myList = new ArrayList();
```

Common Mistakes

- Trying to update **`ArrayList`** values while using an enhanced **`for`** loop

```
for (double score : grades)
{
    if (score < 70)
    {
        score = 70);
    }
}
```

```
Original grades:
81.3 97.5 65.8 100.0 90.0 44.3
Updating grades...
81.3 97.5 70.0 100.0 90.0 70.0
Verification of grades:
81.3 97.5 65.8 100.0 90.0 44.3
```

- Changing the size of an **`ArrayList`** while traversing with an enhanced **`for`** loop

```
for (double score : grades)
{
    if (score < 70)
    {
        grades.add(130.0);
    }
}
```

```
Original grades:
81.3 97.5 65.8 100.0 90.0 44.3
Updating grades...
81.3 97.5 65.8 Exception in thread "main"
java.util.ConcurrentModificationException
        at
java.util.ArrayList$Itr.checkForComodification(ArrayList.ja
va:909)
        at java.util.ArrayList$Itr.next(ArrayList.java:859)
        at Video7P3.main(Video7P3.java:17)
```

- Removing an element from an **`ArrayList`** at the wrong time

```
for (int i = 0; i < grades.size() ; i++)
{
        if (grades.get(i) < 70.0)
        {
                grades.remove(i);
        }
}
```

0	100
1	100
2	60
3	50
4	80
5	70

0	100
1	100
2	50
3	80
4	70

Confusing **Array** With **ArrayList**

```
ArrayList<Character> letters = new ArrayList<Character>(15);
letters[2] = new Character( 'c');
int n = letters.length();
letters[15] = 'c';
```

Can you find the error(s)?

- `[ ]` notation—instead use **`letters.set(2,..)`** or **`letters.add(..)`**
- `.length`—instead use **`.size()`**
- `IndexOutOfBounds`—If you want to add another element, use **`letters.add(15, 'c');`**

Developing Algorithms Using Arrays (ArrayList)

- There are standard algorithms that utilize array traversals to:
  - Determine a minimum or maximum value
  - Compute a sum, average, or mode
  - Determine if at least one element has a particular property
  - Determine if all elements have a particular property
  - Access all consecutive pairs of elements
  - Determine the presence or absence of duplicate elements
  - Determine the number of elements meeting specific criteria
- There are standard array algorithms that utilize traversals to:
  - Shift or rotate elements left or right
  - Reverse the order of the elements

2D Arrays

Consider the following arrays of students' test grades.

```
Student0  =  { 100,    85,    95,    96 }
Student1  =  {  98,   100,   100,    95 }
Student2  =  {  92,   100,    98,   100 }
Student3  =  { 100,    95,    97,    99 }
Student4  =  { 100,   100,   100,    70 }
Student5  =  { 100,    98,    99,    98 }
Student6  =  { 100,    94,   100,    93 }
```

Consider an array of grades where each entry in the array is an array of a student's grades.

```
grades = { Student0, Student1, Student2, Student3, Student4, Student5, Student6 }
```

`grades` is a 2D array because it is an array of arrays.

2D Array Representation of `grades`

```
                Test0          Test1          Test2          Test3
            {
Student0 { 100,             85,            95,            96    },
Student1 {  98,            100,           100,            95    },
Student2 {  92,            100,            98,           100    },
Student3 { 100,             95,            97,            99    },
Student4 { 100,            100,           100,            70    },
Student5 { 100,             98,            99,            98    },
Student6 { 100,             94,           100,            93    }
            }
```

- There are 7 arrays of grades, one for each student.
- In each array there are 4 grades, one for each test.
- This is a rectangular 2D array, because each row array has the same number of entries. *(Non-rectangular 2D arrays are beyond the course's scope.)*

Declaring a 2D Array

Declaring a 2D array is very similar to declaring a 1D array.

```
DataType[]      nameOf1DArray
DataType[][]    nameOf2DArray
```

```
int[][] grades
```

Initializing a 2D Array

Initializing a 2D array is very similar to initializing a 1D array.
You need to know the number of rows (i.e., the number of arrays of arrays).
You need to know the number of columns (i.e., the length of each array row).

```
new DataType[r] [c]
```

```
new int[7][4]
```

Initializing a 2D Array

If you know exactly what the elements of the 2D array are you can initialize the 2D array with a set of initializer lists.

{ {"Alice", "Rob", "Cody" } , {"Robin", "Becky", "Kisha"}

Every initializer list starts and ends with curly brackets. { }

Each row of a 2D array has its own initializer list (i.e., its own { }).

To separate elements of an array, you use a comma. ,

The same goes for separating a row array from another row array. } , {

```
String[][] name ={{"Alice","Rob","Cody"},          //row0
                  {"Robin","Becky","Kisha"}};  //row1
```

size of 2D Array

Because a 2D array has two dimensions, we refer to its size by number of rows along with the number of columns.

r by c **7 by 4**

Because a 2D array is an array of arrays, the number of rows of a 2D array can be determined by calling the **length** attribute of the 2D array to determine the number of arrays in the 2D array.

```
r = grades.length
```

The number of columns of a 2D array is based on the size of each row array. In this course, each row array in a 2D array has the same number of elements. So by convention, we just determine the number of elements in the first row array.

```
c = grades[0].length
```

Accessing elements of a 2D Array

Suppose **Student5**'s **Test2** grade was entered incorrectly. How can we access it to correct our mistake?

```
                Test0          Test1          Test2          Test3
          {
Student0 { 100,            85,            95,            96    },
Student1 {  98,           100,           100,            95    },
Student2 {  92,           100,            98,           100    },
Student3 { 100,            95,            97,            99    },
Student4 { 100,           100,           100,            70    },
Student5 { 100,            98,            99,            98    },
Student6 { 100,            94,           100,            93    }
          }
```

Like a 1D array, you access an array element using bracket notation and the **index** of the element's location in both dimensions.

```
grades[5][2]
grades[grades.length - 1][grades[0].length - 1]
```

Updating elements of a 2D Array

Suppose **Student5**'s **Test2** grade was entered incorrectly. It is supposed to be 100 instead of 99. How can we update it?

```
                Test0        Test1        Test2        Test3
          {
Student0 { 100,           85,          95,          96   },
Student1 {  98,          100,         100,          95   },
Student2 {  92,          100,          98,         100   },
Student3 { 100,           95,          97,          99   },
Student4 { 100,          100,         100,          70   },
Student5 { 100,           98,         100,          98   },
Student6 { 100,           94,         100,          93   }
          }
```

```
grades[5][2]= grades[5][2] + 1;
```

Traversing 2D Arrays

Row-Major Order vs. Column-Major Order

```
String[][] grid = new String[4][5];    // populated elsewhere…
```

Printing contents in row-major order

	0	1	2	3	4
0	"A"	"B"	"C"	"D"	"E"
1	"F"	"G"	"H"	"I"	"J"
2	"K"	"L"	"M"	"N"	"O"
3	"P"	"Q"	"R"	"S"	"T"

```
ABCDEFGHIJKLMNOPQRST
```

Printing contents in column-major order

	0	1	2	3	4
0	"A"	"B"	"C"	"D"	"E"
1	"F"	"G"	"H"	"I"	"J"
2	"K"	"L"	"M"	"N"	"O"
3	"P"	"Q"	"R"	"S"	"T"

```
AFKPBGLQCHMRDINSEJOT
```

Traversing 2D Arrays

Row-major

What if we don't know the dimensions of the 2D array?

```
public static void printArray(String[][] grid)
{
    for (int row = 0; row < grid.length; row++)
    {
        for (int col = 0; col < grid[0].length; col++)
        {
            System.out.print(grid[row][col]);
        }
        System.out.println();
    }
}
```

Column-major

Traversing 2D Arrays

How can we modify the code for **column-major** order?

```
public static void printArray(String[][] grid)
{
    for (int col = 0; col < grid[0].length; col++)
    {
        for (int row = 0; row < grid.length; row++)
        {
            System.out.print(grid[row][col]);
        }
        System.out.print(" ");
    }
    System.out.println();
}
```

```
AFKP BGLQ CHMR DINS EJOT
_
```

UNFORTUNATELY… This means you can't easily use enhanced `for` loops for column-major order traversal ☹

grid =

	0	1	2	3	4
0	"A"	"B"	"C"	"D"	"E"
1	"F"	"G"	"H"	"I"	"J"
2	"K"	"L"	"M"	"N"	"O"
3	"P"	"Q"	"R"	"S"	"T"

Traversing 2D Arrays

What about using enhanced **`for`** loops (**`for-each`** loops)?

```
public static void printArray(String[][] grid)
{
    for (String[] row : grid)
    {
        for (String letter : row)
        {
            System.out.print(letter);
        }
        System.out.println();
    }
}
```

UNFORTUNATELY… This means you can't easily use enhanced `for` loops for column-major order traversal ☹