innovate achieve lead

BITS Pilani

Pilani Campus

# Object Oriented Programming

## Arrays

# Introduction to Arrays

- An *array* is a data structure used to process a collection of data that is all of the same type
  - An array behaves like a numbered list of variables with a uniform naming mechanism
  - It has a part that does not change:
    - the name of the array
  - It has a part that can change:
    - an integer in square brackets
  - For example, given five scores:

```
score[0],score[1],score[2],score[3],score[4]
```

# Creating and Accessing Arrays

- An array that behaves like a collection of variables, all of type **double**, can be created using one statement

```
double[] score = new double[5];
```

- Or using two statements:

```
double[] score;
score = new double[5];
```

  - The first statement declares the variable **score** to be of the array type **double[]** (an array of doubles)
  - The second statement creates an array with five numbered variables of type **double** and makes the variable **score** a name for the array

# Creating and Accessing Arrays

- The individual variables that together make up the array are called *indexed variables*
  - They can also be called *subscripted variables* or *elements* of the array
  - The number in square brackets is called an *index* or *subscript*
  - The number of indexed variables in an array is called the *length* or *size* of the array
  - In Java *indices must be numbered starting with 0, and nothing else*

```
score[0], score[1], score[2], score[3], score[4]
```

# Creating and Accessing Arrays

- When an array is created, the length of the array is given in square brackets after the array type

- The indexed variables are then numbered starting with **0**, and ending with the integer that is *one less than the length of the array*

- The declaration

```
double[] score = new double[5];
```

results in the 5 elements

```
score[0], score[1], score[2], score[3], score[4]
```

# Creating and Accessing Arrays

```
double[] score = new double[5];
```

- A variable may be used in place of the integer (i.e., in place of the integer *5* above)
  - The value of this variable can then be read from the keyboard
  - This enables the size of the array to be determined when the program is run

```
double[] score = new double[count];
```

- An array can have indexed variables of any type, including any class type

- All of the indexed variables in a single array must be of the same type, called the *base type* of the array

# Declaring and Creating an Array

- An array is declared and created in almost the same way that objects are declared and created:

```
BaseType[] ArrayName = new BaseType[size];
```

  - The *size* may be given as an expression that evaluates to a nonnegative integer, for example, an `int` variable

```
char[] line = new char[80];
double[] reading = new double[count];
Person[] specimen = new Person[100];
```

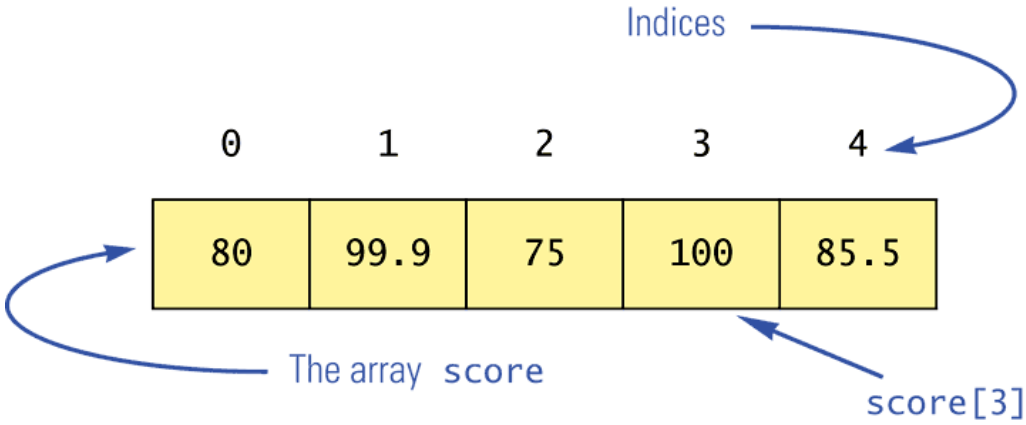
# Referring to Arrays and Array Elements

- Each array element can be used just like any other single variable by referring to it using an indexed expression: `score[0]`
- The array itself (i.e., the entire collection of indexed variables) can be referred to using the array name (without any square brackets): `score`
- An array index can be computed when a program is run
  - It may be represented by a variable: `score[index]`
  - It may be represented by an expression that evaluates to a suitable integer: `score[next + 1]`

# Using the `score` Array in a Program

- The `for` loop is ideally suited for performing array manipulations:

```
for (int index = 0; index < 5; index++)
  System.out.println(score[index]
          + " differs from max by "
          + (max - score[index]) );
```



# Three Ways to Use Square Brackets [ ] with an Array Name



- Square brackets can be used to create a type name:

```
double[] score;
```

- Square brackets can be used with an integer value as part of the special syntax Java uses to create a new array:

```
score = new double[5];
```

- Square brackets can be used to name an indexed variable of an array:

```
max = score[0];
```

# The `length` Instance Variable

- **An array is considered to be an object**
- Since other objects can have instance variables, so can arrays
- Every array has exactly one instance variable named ***`length`***
  - When an array is created, the instance variable **`length`** is automatically set equal to its size
  - The value of **`length`** cannot be changed (other than by creating an entirely new array using **`new`**)

```
double[] score = new double[5];
```

  - Given **`score`** above, **`score.length`** has a value of 5

# Pitfall: Array Index Out of Bounds

- Array indices always start with 0, and always end with the integer that is one less than the size of the array
  - The most common programming error made when using arrays is attempting to use a nonexistent array index
- When an index expression evaluates to some value other than those allowed by the array declaration, the index is said to be *out of bounds*
  - An out of bounds index will cause a program to terminate with a run-time error message
  - Array indices get out of bounds most commonly at the *first* or *last* iteration of a loop that processes the array: Be sure to test for this!

# Initializing Arrays

- An array can be initialized when it is declared
  - Values for the indexed variables are enclosed in braces, and separated by commas
  - The array size is automatically set to the number of values in the braces

```
int[] age = {2, 12, 1};
```

  - Given **`age`** above, **`age.length`** has a value of 3

# Initializing Arrays

- Another way of initializing an array is by using a **`for`** loop

```
double[] reading = new double[100];
for (int index = 0; index < reading.length; index++){
  reading[index] = 42.0;
}
```

- If the elements of an array are not initialized explicitly, they will automatically be initialized to the default value for their base type

# An Array of Characters Is Not a String

- An array of characters is conceptually a list of characters, and so is conceptually like a string
- However, an array of characters is not an object of the class **`String`**

```
char[] a = {'A', 'B', 'C'};
String s = a; //Illegal!
```

- An array of characters can be converted to an object of type **`String`**, however

(continued)

# An Array of Characters Is Not a String

- The class **`String`** has a constructor that has a single parameter of type **`char[]`**

```
String s = new String(a);
```

  - The object **`s`** will have the same sequence of characters as the entire array **`a`** (**`"ABC"`**), but is an *independent* copy

- Another **`String`** constructor uses a subrange of a character array instead

```
String s2 = new String(a,0,2);
```

  - Given **`a`** as before, the new string object is **`"AB"`**

(continued)

# An Array of Characters Is Not a String

- An array of characters does have some things in common with **`String`** objects
  - For example, an array of characters can be output using **`println`**

```
System.out.println(a);
```

  - Given **`a`** as before, this would produce the output

```
ABC
```

# Arrays and References

- Like class types, a variable of an array type holds a *reference*
  - Arrays are objects
  - A variable of an array type holds the address of where the array object is stored in memory
  - Array types are (usually) considered to be class types

# Arrays are Objects

- An array can be viewed as a collection of indexed variables
- An array can also be viewed as a single item whose value is a collection of values of a base type
  - An array variable names the array as a single item
    ```
    double[] a;
    ```
  - A **new** expression creates an array object and stores the object in memory
    ```
    new double[10]
    ```
  - An assignment statement places a reference to the memory address of an array object in the array variable
    ```
    a = new double[10];
    ```
  - The previous steps can be combined into one statement
    ```
    double[] a = new double[10];
    ```

(continued)

# Arrays with a Class Base Type

- The base type of an array can be a class type

```
Date[] holidayList = new Date[20];
```

- The above example creates 20 indexed reference variables of type `Date`. It **does not** create 20 objects of the class `Date`
  - Each of these indexed variables are automatically initialized to `null`
  - Any attempt to reference any them at this point would result in a "null pointer exception" error message

(continued)

# Arrays with a Class Base Type

- Like any other object, each of the indexed variables requires a separate invocation of a constructor using **new** (singly, or perhaps using a **for** loop) to create an object to reference

```
holidayList[0] = new Date();
                . . .
holidayList[19] = new Date();
```

OR

```
for (int i = 0; i < holidayList.length; i++)
  holidayList[i] = new Date();
```

- Each of the indexed variables can now be referenced since each holds the memory address of a **Date** object

# Array Parameters

- Both array indexed variables and entire arrays can be used as arguments to methods
  - An indexed variable can be an argument to a method in exactly the same way that any variable of the array base type can be an argument

# Array Parameters

```
double n = 0.0;
double[] a = {2.3, 4.5, 6.7, 8.9};
int i = 2;
```

Given the method declaration

```
public void myMethod (double x)
```

then all of the following are legal:

```
myMethod(n);          //n evaluates to 0.0
myMethod(a[3]);       //a[3] evaluates to 8.9
myMethod(a[i]);       //i evaluates to 2,
                      //a[2] evaluates to 6.7
```

# Array Parameters

- An argument to a method may be an entire array
- Array arguments behave like objects of a class
  - Therefore, a method can change the values stored in the indexed variables of an array argument
- A method with an array parameter must specify the base type of the array only

```
BaseType[]
```

  - It does not specify the length of the array

# Array Parameters

The following method, **doubleElements**, specifies an array of **double** as its single argument:

```
public class SampleClass{
  public static void doubleElements(double[] a){
    int i;
    for (i = 0; i < a.length; i++)
      a[i] = a[i] * 2;
    . . .
  }
. . .
}
```

(continued)

# Array Parameters

- Arrays of double may be defined as follows:

```
double[] a = new double[10];
double[] b = new double[30];
```

- Given the arrays above, the method `doubleElements` from class `SampleClass` can be invoked as follows:

```
SampleClass.doubleElements(a);
SampleClass.doubleElements(b);
```

  - Note that no square brackets are used when an entire array is given as an argument
  - Note also that a method that specifies an array for a parameter can take an array of any length as an argument

# Use of = and == with Arrays

- Because an array variable contains the memory address of the array it names (it's a reference), the assignment operator (=) only copies this memory address
  - It does not copy the values of each indexed variable
  - Using the assignment operator `b = a;` will make two array variables be different names for the same array
  - The memory address in `a` is now the same as the memory address in `b`: They reference the same array

(continued)

# Use of = and == with Arrays

- A **for** loop is usually used to make two different arrays have the same values in each indexed position:

```
for (int i = 0;(i < a.length) && (i < b.length); i++){
  b[i] = a[i];
}
```

  - Note that the above code will **not** make **b** an exact copy of **a**, unless **a** and **b** have the same length

(continued)

# Use of = and == with Arrays

- For the same reason, the equality operator **(==)** only tests two arrays to see if they are stored in the same location in memory
  - **`(a == b)`** does not test two arrays to see if they contain the same values
  - The result of the above **`boolean`** expression will be **`true`** if **`a`** and **`b`** share the same memory address (and, therefore, reference the same array), and **`false`** otherwise

(continued)

# Use of = and == with Arrays

- In the same way that an **`equals`** method can be defined for a class, an ***`equalsArray`*** method can be defined for a type of array
  - This is how two arrays must be tested to see if they contain the same elements

# Use of = and == with Arrays

```
public static boolean equalsArray(int[] a,
                                  int[] b){

  if (a.length != b.length)  return false;
  else{
    int i = 0;
    while (i < a.length){
      if (a[i] != b[i])
        return false;
      i++;
    }
  }
  return true;
}
```

# Arguments for the Method **`main`**

- The heading for the **`main`** method of a program has a parameter for an array of **`String`**
  - It is usually called ***`args`*** by convention

```
public static void main(String[] args)
```

  - Note that since ***`args`*** is a parameter, it could be replaced by any other non-keyword identifier
- If a Java program is run without giving an argument to **`main`**, then a default empty array of strings is automatically provided

# Arguments for the Method `main`

- If a program requires that the `main` method be provided an array of strings argument, each element must be provided from the command line when the program is run

```
java SomeProgram Hi ! there
```

  - This will set `args[0]` to "Hi", `args[1]` to "!", and `args[2]` to "there"
  - It will also set `args.length` to 3

# Methods That Return an Array

- In Java, a method may also return an array
  - The return type is specified in the same way that an array parameter is specified. This method returns an array of int

```
public static int[] incrementArray(int[] a,
                                   int increment){
   int[] temp = new int[a.length];
   for (int i = 0; i < a.length; i++)
     temp[i] = a[i] + increment;
   return temp;
 }
```

# Privacy Leaks with Array Instance Variables

- If an accessor method does return the contents of an array, special care must be taken
  - Just as when an accessor returns a reference to any private object

```
public double[] getArray(){
   return anArray;
}
```

  - The example above will result in a *privacy leak.*
  - Why is this so?

# Privacy Leaks with Array Instance Variables

- The previous accessor method would simply return a reference to the array **`anArray`** itself

- Instead, an accessor method should return a reference to a *deep copy* of the private array object
  - Below, **`a`** is an array which is an instance variable of the class containing the **`getArray`** method

```
public double[] getArray(){
   double[] temp = new double[a.length];
   for (int i = 0; i < a.length; i++)
     temp[i] = a[i];
   return temp
}
```

# Privacy Leaks with Array Instance Variables

- If a private instance variable is an array that has a class as its base type, then copies must be made of each class object in the array when the array is copied. Here `b` is an array of class types and an instance variable of the class containing the `getArray` method

```
public ClassType[] getArray(){
  ClassType[] temp = new ClassType[b.length];
  for (int i = 0; i < b.length; i++)
    temp[i] = new ClassType(b[i]);
  return temp;
}
```

# Multidimensional Arrays

- It is sometimes useful to have an array with more than one index
- Multidimensional arrays are declared and created in basically the same way as one-dimensional arrays
  - You simply use as many square brackets as there are indices
  - Each index must be enclosed in its own brackets

```
double[][]table = new double[100][10];
int[][][] figure = new int[10][20][30];
Person[][] = new Person[10][100];
```

# Multidimensional Arrays

- Multidimensional arrays may have any number of indices, but perhaps the most common number is two
  - Two-dimensional array can be visualized as a two-dimensional display with the first index giving the row, and the second index giving the column

```
char[][] a = new char[5][12];
```

  - Note that, like a one-dimensional array, each element of a multidimensional array is just a variable of the base type (in this case, **`char`**)

# Multidimensional Arrays

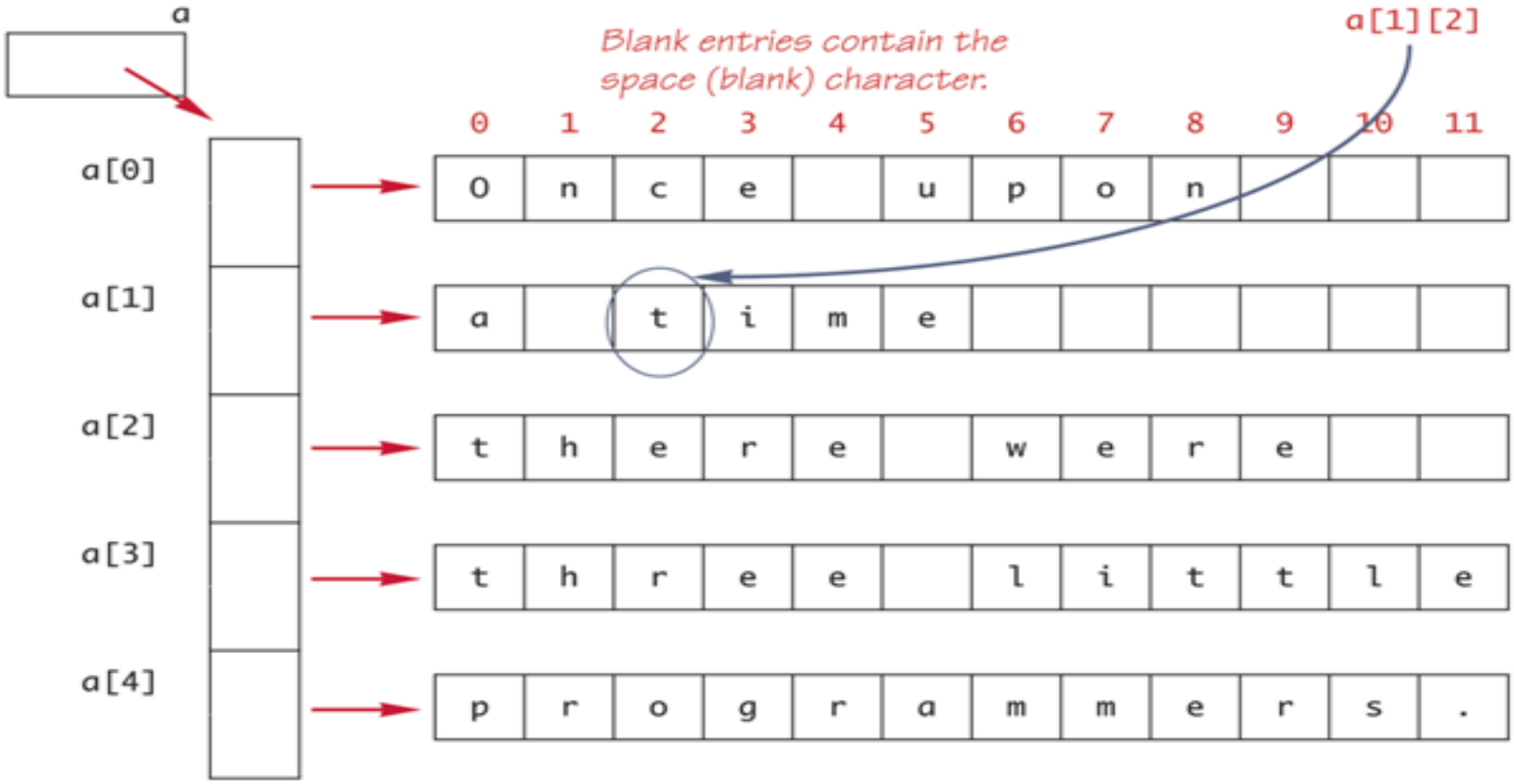
- In Java, a two-dimensional array, such as **`a`**, is actually an array of arrays
  - The array **`a`** contains a reference to a one-dimensional array of size 5 with a base type of **`char[]`**
  - Each indexed variable (**`a[0]`**, **`a[1]`**, etc.) contains a reference to a one-dimensional array of size 12, also with a base type of **`char[]`**
- A three-dimensional array is an array of arrays of arrays, and so forth for higher dimensions

# Two-Dimensional Array as an Array of Arrays (Part 1 of 2)



```
char[][] a = new char[5][12];
```

*Code that fills the array is not shown.*



# Two-Dimensional Array as an Array of Arrays (Part 2 of 2)



*We will see that these can and should be replaced with expressions involving the* `length` *instance variable.*

```
int row, column;
for (row = 0; row < 5; row++)
{
    for (column = 0; column < 12; column++)
       System.out.print(a[row][column]);
    System.out.println();
}
```

*Produces the following output:*

```
Once upon
a time
there were
three little
programmers.
```

# Using the `length` Instance Variable

```
char[][] page = new char[30][100];
```

- The instance variable **`length`** does not give the total number of indexed variables in a two-dimensional array
  - Because a two-dimensional array is actually an array of arrays, the instance variable **`length`** gives the number of first indices (or "rows") in the array
    - **`page.length`** is equal to 30
  - For the same reason, the number of second indices (or "columns") for a given "row" is given by referencing **`length`** for that *"row" variable*
    - **`page[0].length`** is equal to 100

# Using the `length` Instance Variable

- The following program demonstrates how a nested **`for`** loop can be used to process a two-dimensional array
  - Note how each **`length`** instance variable is used

```
int row, column;
for (row = 0; row < page.length; row++)
  for (column = 0; column < page[row].length;
                                        column++)
    page[row][column] = 'Z';
```

# Multidimensional Array Parameters and Returned Values



- Methods may have multidimensional array parameters
  - They are specified in a way similar to one-dimensional arrays
  - They use the same number of sets of square brackets as they have dimensions

```
public void myMethod(int[][] a)
{ . . . }
```

  - The parameter **a** is a two-dimensional array

# Multidimensional Array Parameters and Returned Values

- Methods may have a multidimensional array type as their return type
  - They use the same kind of type specification as for a multidimensional array parameter

```
public double[][] aMethod()
{ . . . }
```

  - The method **aMethod** returns an array of **double**

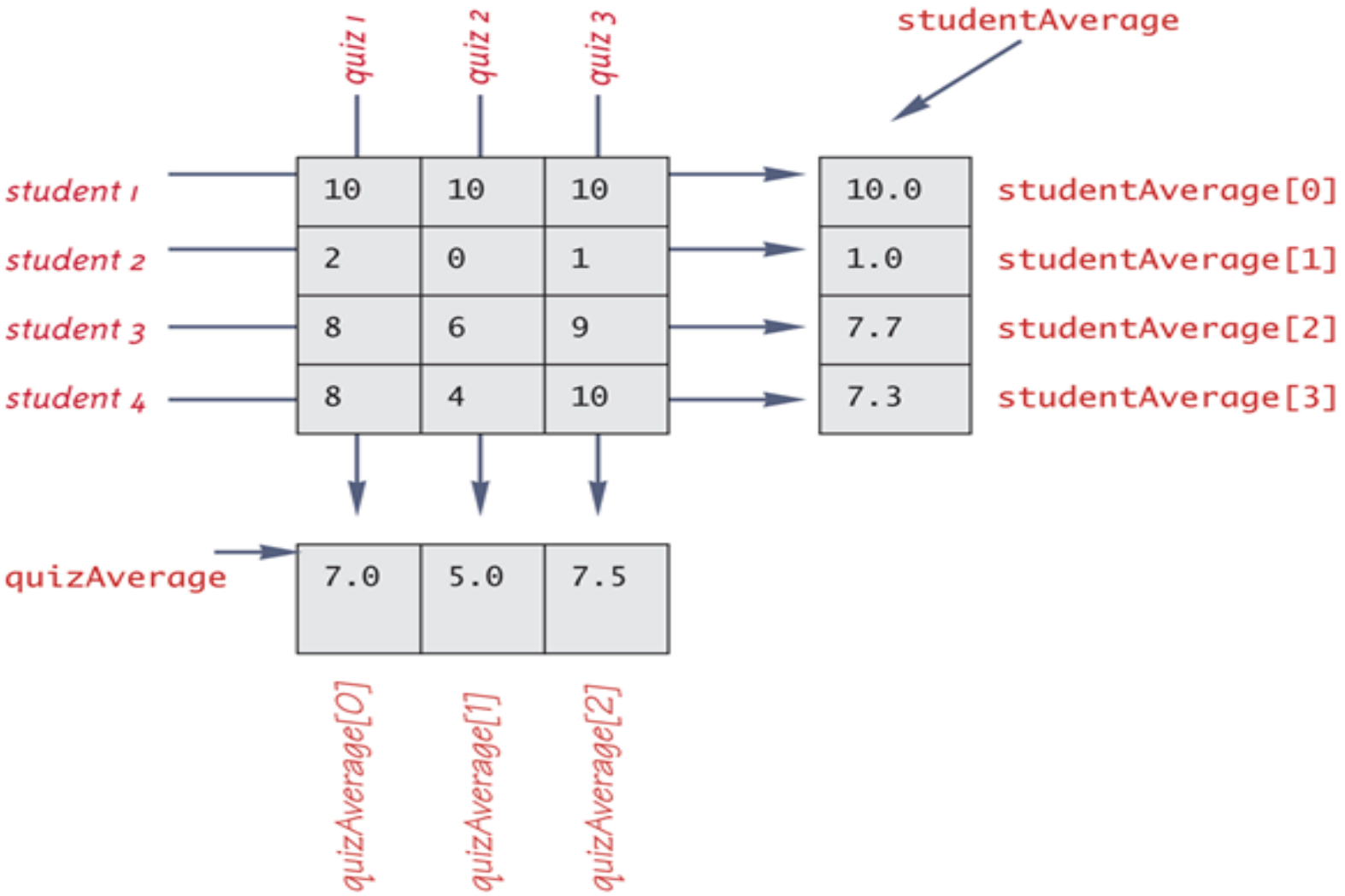
# A Grade Book Class

- As an example of using arrays in a program, a class **`GradeBook`** is used to process quiz scores
- Objects of this class have three instance variables
  - **`grade`**: a two-dimensional array that records the grade of each student on each quiz
  - **`studentAverage`**: an array used to record the average quiz score for each student
  - **`quizAverage`**: an array used to record the average score for each quiz

# A Grade Book Class

- The score that student 1 received on quiz number 3 is recorded in **`grade[0][2]`**
- The average quiz grade for student 2 is recorded in **`studentAverage[1]`**
- The average score for quiz 3 is recorded in **`quizAverage[2]`**
- Note the relationship among the three arrays

# The 2-Dimensional Array **grade**



# THANK YOU