

# Lecture - Programming with arrays

## Partially filled arrays

The size needed for an array is not always known when the first element is written to the array. Even dynamic allocation of arrays cannot always tell us how big the array should be.

An obvious (but bad) way to handle this is to make the array grow every time it is written to.

```
class PartiallyFilledArrays {
    public static void main (String[] args) {
        int max = 100;
        int array[] = {}; array size 0

        // Record how many elements are copied in memory, in total
        int elementsReallocated = 0;

        // Fill up the array, in order
        for (int i = 0; i < max; i++) {
            // If the array isn't big enough for element i,
            // then extend it until it is big enough.
            if (i > array.length - 1) {
                int[] tmp = new int [i+1];
                for (int j = 0; j < array.length; j++) {
                    tmp[j] = array[j];
                    elementsReallocated++;
                }
                array = tmp;
            }
            array[i] = i * i;
        }
        System.out.println ("To fill an array of " + max
            + " elements required " + elementsReallocated
            + " reallocations,");
        System.out.println("which is about "
            + elementsReallocated/max + " each.");
    }
}
```

$$\frac{(1+99) \times 100}{2} = 4950$$

**Exercise:** Vary `max` above, and see how the number of reallocations varies. If you like maths, try to calculate the total number for a general value of `max`.

A common way to handle this is to declare the array to be of the *largest* size that the program could possibly need.

Care must then be taken to keep track of how much of the array is *actually* used



An indexed variable that has not been given a meaningful value **must never be referenced**. Its value is undefined, and can lead to undefined behaviour of the programming. This leads to intermittent errors, which are really hard to debug.

This is different from referring to a `null` object variable. That will immediately cause the code to crash. That should also never happen in production code, but at least it is easier to debug.

## Keeping track of what is used

A variable can be used to keep track of how many elements are currently stored in an array

For example, given the variable `count`, the elements of the array `someArray` will range from positions `someArray[0]` to `someArray[count - 1]`.

Note that the variable `count` will be used to process the partially filled array instead of `someArray.length`.

Note also that this variable ( `count` ) must be an argument to any method that manipulates the partially filled array

## A growing array class

It is possible to group `count` together with the array in a class, so there is no need to pass `count` along with the partially filled array.

```
class GrowingArray {
    static public void main (String[] args) {
        int max = 10;
        varArray array = new varArray();

        // Fill up the array, in order
        for (int i = 0; i < max; i++) {
            array.put(i, i*i);
        }

        System.out.println ("To fill an array of " + max
            + " elements required " + varArray.elementsReallocated()
            + " reallocations,");
        System.out.println("which is about "
            + (double)varArray.elementsReallocated()/max + " each.");
    }
}

class varArray {
    static int realloc = 0;
    static int elementsReallocated () {
        return realloc;
    }
}
```

```

int [] array;
public int length;

varArray () {
    length = 0;
    array = new int[length];
}

void put (int idx, int val) {
    if (idx >= length) {
        // What happens if you change this to
        length=Math.max(idx+1, 2*length);
        //length = idx + 1;
        int[] tmp = new int [length];
        for (int j = 0; j < array.length; j++) {
            tmp[j] = array[j];
            realloc++;
        }
        array = tmp;
    }
}

int get (int idx) {
    if (idx < length) {
        return array[idx];
    } else {
        System.err.println("Accessing element "
            + idx + " of array of size " + length);
        System.exit(1);
        // never called
        return (0);
    }
}
}

```

可以選擇直接加1或2倍  
 This one only need 15 reallocation.  
 if use this one need 45 reallocations.




In some languages like C++, it is possible to overload operators as well as regular methods. In those languages, it is possible to write classes that behave just like arrays, even using `someArray[idx]` for access.

**Exercise:** If we can have partly-filled arrays, it is possible to extend them more efficiently. Modify the above code, as indicated in the comment, and notice that, by doubling the array size each time it grows, the number of reallocations per element remains no more than 2. (When is it highest?) This is a particularly efficient yet simple data structure for storing growing arrays.

# The "for each" loop


The standard Java libraries include a number of collection classes: classes whose objects store a collection of values.

 Note: we will cover more about collection classes in later lessons.

Ordinary counting `for` loops cannot cycle through the elements in a collection object.

Unlike array elements, collection object elements are not normally associated with indices.

However, there is a new kind of `for` loop, first available in Java 5.0, called a *for-each loop* or *enhanced for loop*. This kind of loop can cycle through each element in a collection even if the elements are not indexed.

 Note: We have covered a basic syntax of Enhanced For Loop in [Week 2](#)

It can also cycle through an array.

The syntax for a for-each loop statement used with an array is

```
for (ArrayBaseType variableName : arrayName)
    Statement;
```

The above for-each line should be read as "for each `variableName` in `arrayName` do the following:"

- Note that `variableName` must be declared *within* the for-each loop, not before.
- Note also that a colon (not a semicolon) is used after `variableName`.



`variableName` is just a regular reference variable. If you assign to it, you make it refer to a different object, rather than assigning to the element of the array.

**Exercise:** In the following code, replace the regular for loop by a for-each loop. Which one behaves as you would expect, and which one is affected by the warning above?

```
class ArrayIterationDemo {
    public static void main (String[] args) {
        double a[] = new double [] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

        for (int i = 0; i < a.length; i++) {
            a[i] += 1;
        }

        for (int i = 0; i < a.length; i++) {
            System.out.print (" " + a[i]);
        }
    }
}
```

```
}  
System.out.println("");  
}  
}
```

```
9  
10 for(double a: arr){  
11     a = a+5;  
12     System.out.println(a);  
13 }  
14 for (int i = 0; i < arr.length; i++) {  
15     System.out.print (" " + arr[i]);  
16 }
```

6.0  
7.0  
8.0  
9.0  
10.0  
11.0  
12.0  
13.0  
14.0

→ it did not get reflect to the original array.

because we do this to the primitive data type.  
but the array is called by reference.  
we didn't change the array.

$\tau_{\text{arg}} = \text{variable number of argument.}$

## Methods with a variable number of parameters

Starting with Java 5.0, methods can be defined that take any number of arguments.

Essentially, it is implemented by taking in an array as argument, but the job of placing values in the array is done automatically. The values for the array are given as arguments. Java automatically creates an array and places the arguments in the array. Note that arguments corresponding to regular parameters are handled in the usual way.

Such a method has as the last item on its parameter list a vararg specification of the form:

```
Type... ArrayName
```

Note the three dots called an *ellipsis* that must be included as part of the vararg specification syntax.

Following the arguments for regular parameters are any number of arguments of the type given in the vararg specification. These arguments are automatically placed in an array. This array can be used in the method definition. Note that a vararg specification allows any number of arguments, *including zero*.



Note that Java does not have "default parameters", like languages such as C++ and Python. Those are values that are actually always passed to the method, but just not explicitly mentioned in the calling syntax; the compiler or interpreter adds the default values that were not specified in the call.

Here is an example of a function that takes a variable number of argument.

```
class MultipleParameters {
    /**
     Returns the largest of any number of int values.
    */
    public static int max (int... arg) {
        int largest = Integer.MIN_VALUE;
        for (int a : arg)
            if (a > largest)
                largest = a;
        return largest;
    }

    public static void main (String[] arg) {
        System.out.println(max (1, 2) + " " + max(4, 6, 3) + " " + max());
    }
}
```

$\Rightarrow 2 \quad 6 \quad -2147483648$

One standard function that uses variable numbers of parameters is `System.out.printf`. The syntax is

```
System.out.printf(format, argument1, argument2, ..., argumentN);
```

Unlike `System.out.print`, it does not require the arguments to be concatenated into a single string. The `format` string argument specifies how to convert each argument to a string, and what other characters to put between them.

**Advanced:** The arguments do not need to be of the same type. So how can this function use the mechanism we have just discussed? In a later lecture, you will learn about polymorphism, in which an object can have more than one type. All class objects have type `Object` in addition to the class they are defined as. That means that `printf` has a declaration ending with `Object... args`. It uses the `format` to work out the "true" type of each of its arguments, and that lets it format them correctly.

# Privacy leaks with array member variables

Arrays are objects, like class objects. Just as there were risks of privacy leaks with object member variables (static or instance variables), there are risks with array member variables.

If an accessor method does return the contents of an array, special care must be taken

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

The example above will result in a privacy leak. It would simply return a reference to `anArray`. The code `var.getArray()[0] = 1;` will write to `anArray`, which may have been private.

Instead, an accessor method should usually return a reference to a *deep copy* of the private array object.

*deepCopy ↔ shallowCopy*

```
class DeepCopy {  
    double[] anArray = {};  
    public double [] getArray ()  
    {  
        double [] tmp = new double [anArray.length];  
        for (int i = 0; i < anArray.length; i++)  
            tmp[i] = anArray[i];  
        return tmp;  
    }  
  
    public static void main (String[] args) {  
        // Write your own test here.  
    }  
}
```

If a private instance variable is an array that has a class as its base type, the n-copies must be made of each class object in the array when the array is copied:

```
class ArrayCopy {  
    ClassType[] anArray = {};  
    public ClassType [] getArray ()  
    {  
        ClassType [] tmp = new ClassType [anArray.length];  
        for (int i = 0; i < anArray.length; i++)  
            tmp[i] = new ClassType(anArray[i]);  
        return tmp;  
    }  
  
    public static void main (String[] args) {  
        // Write your own test here.  
    }  
}
```



```
    }  
}  
  
class ClassType {  
    // Copy constructor.  
    // Add a member variable to ClassType, and update this copy constructor  
    ClassType(ClassType c) {  
    }  
}
```

# Sorting an array

Sorting is one of the most basic tasks in computer science.

An "in place" sort method takes in an array parameter `a`, and rearranges the elements in `a`, so that after the method call is finished, the elements of `a` are sorted in ascending order

- An alternative is to return a new array that contains the elements of `a` in ascending order, without changing the array `a`.

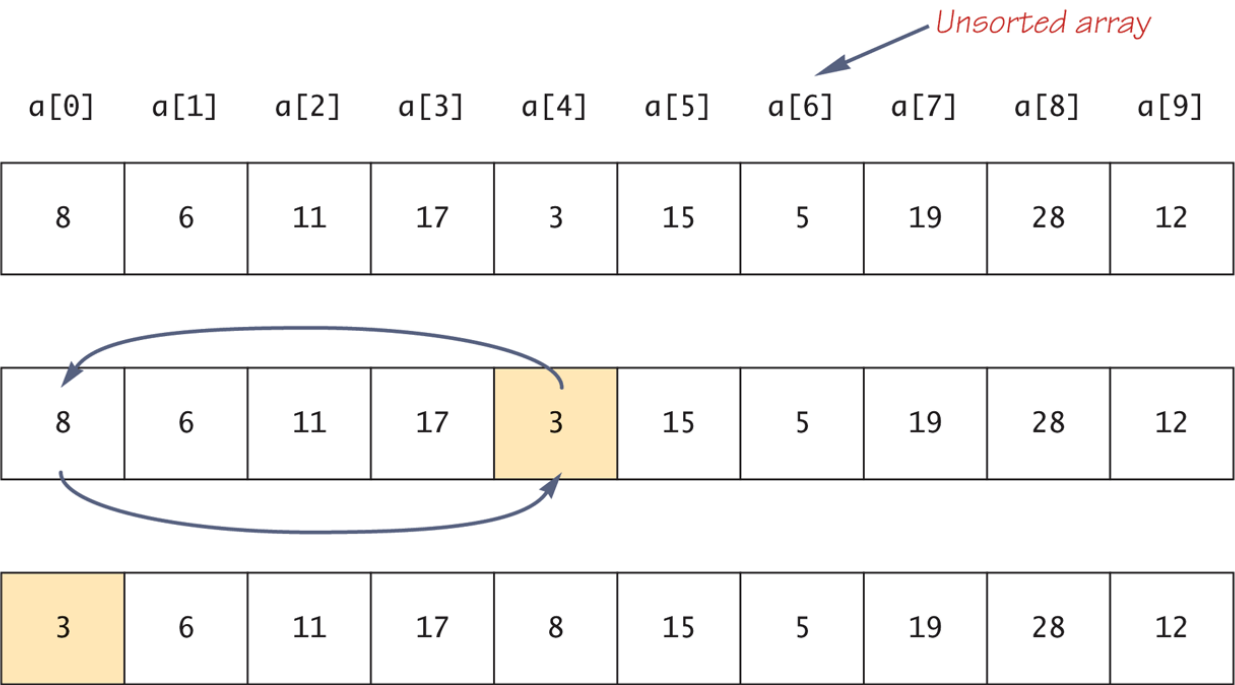
A *selection sort* accomplishes this by using the following algorithm:

```
for (int index = 0; index < count; index++)  
    Place the indexth smallest element of a in output[index]
```

(Selection sort is easy to understand and code, but there are much more efficient algorithms for sorting.)

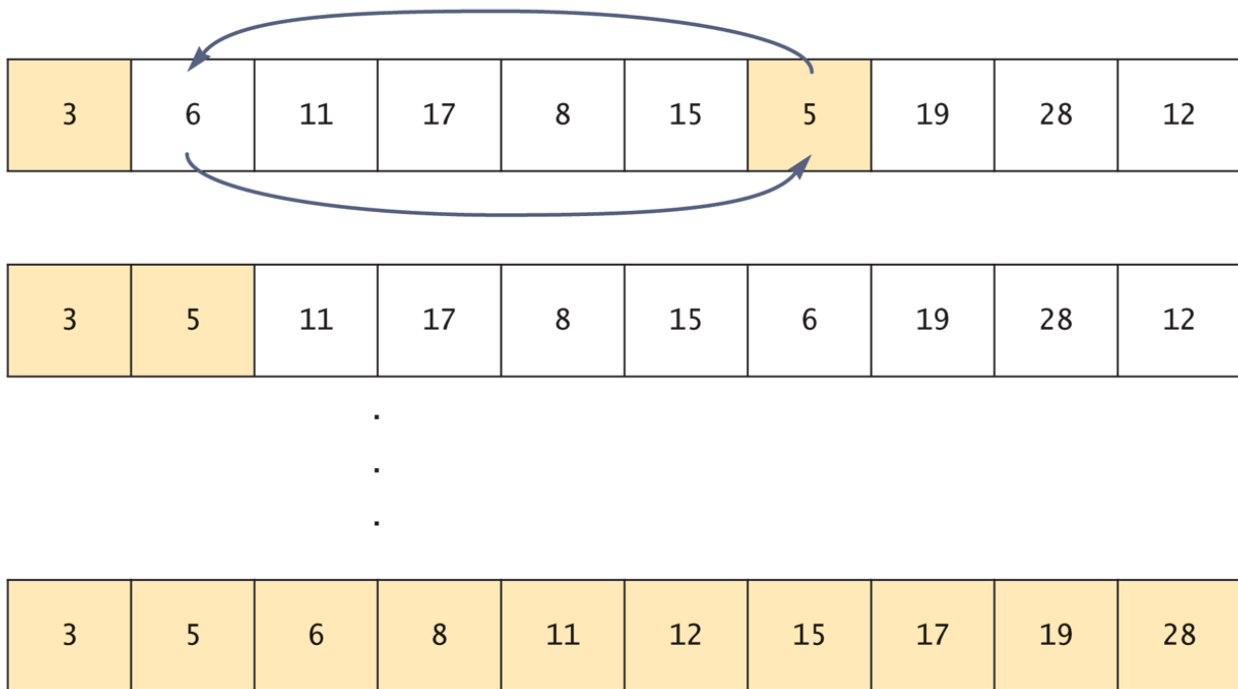
It works as follows:

**Display 6.10    Selection Sort**



(continued)

## Display 6.10 Selection Sort



```
public class Main
{
    /**
     Precondition: count <= a.length;
     The first count indexed variables have
     values.
     Action: Sorts a so that a[0] <= a[1] <=
     ... <= a[count - 1].
     */
    public static void sort(double[] a, int count)
    {
        int index, indexOfNextSmallest;
        for (index = 0; index < count - 1; index++)
        {
            //Place the correct value in a[index]:
            indexOfNextSmallest =
                indexOfSmallest(index, a, count);
            interchange(index, indexOfNextSmallest, a);
            //a[0]<=a[1]<=...<=a[index] and these are
            //the smallest of the original array
            //elements. The remaining positions contain
            //the rest of the original array elements.
        }
    }

    /**
     Returns the index of the smallest value among
     a[startIndex], a[startIndex+1], ...
     a[numberUsed - 1]
     */
    private static int indexOfSmallest(int startIndex, double[] a, int count)
    {
        double min = a[startIndex];
```

```
import java.util.Arrays;
import java.util.Collections;
public class Main
{
    public static void main (String[] arg) {
        int [] a = {6,2,3,6,2,3,4,1};
        Integer[] arr = new Integer[a.length];

        for (int i=0; i<a.length; i++){
            arr[i] = Integer.valueOf(a[i]);
        }
        Arrays.sort(arr, Collections.reverseOrder());
        for (int i=0; i<arr.length; i++){
            System.out.println(arr[i] + " ");
        }

        // sort (a, a.length);
        Arrays.sort(a);

        for (int element : a)
            System.out.print(" " + element);
        System.out.println("");
    }
}
```

*need non-primitive type.*

*to sort the array a*

```

int indexOfMin = startIndex;
int index;
for (index = startIndex + 1;
    index < count; index++)
if (a[index] < min)
{
    min = a[index];
    indexOfMin = index;
    //min is smallest of a[startIndex] through
    //a[index]
}
return indexOfMin;
}

/**
Precondition: i and j are legal indices for
the array a.
Postcondition: Values of a[i] and a[j] have
been interchanged.
*/
private static void interchange(int i, int j, double[] a)
{
    double temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp; //original value of a[i]
}

public static void main (String[] arg) {
    double[] a = {6,2,3,6,2,3,4,1};

    sort (a, a.length);

    for (double element : a)
        System.out.print(" " + element);
    System.out.println("");
}
}

```

**Exercise:** Modify the code to sort the elements of `a` in descending order

**Exercise:** Modify the code to return a sorted copy of `a`, leaving the original `a` unchanged.

**Exercise:** Count the number of times two numbers are compared in `indexOfSmallest`.

## Example of selection sort

An error occurred.

Try watching this video on [www.youtube.com](https://www.youtube.com), or enable JavaScript if it is disabled in your browser.



Example of more efficient sort: Quicksort

An error occurred.

---

Try watching [this video on www.youtube.com](https://www.youtube.com), or enable JavaScript if it is disabled in your browser.

---

# Multidimensional arrays

It is sometimes useful to have an array with more than one index. These "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 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

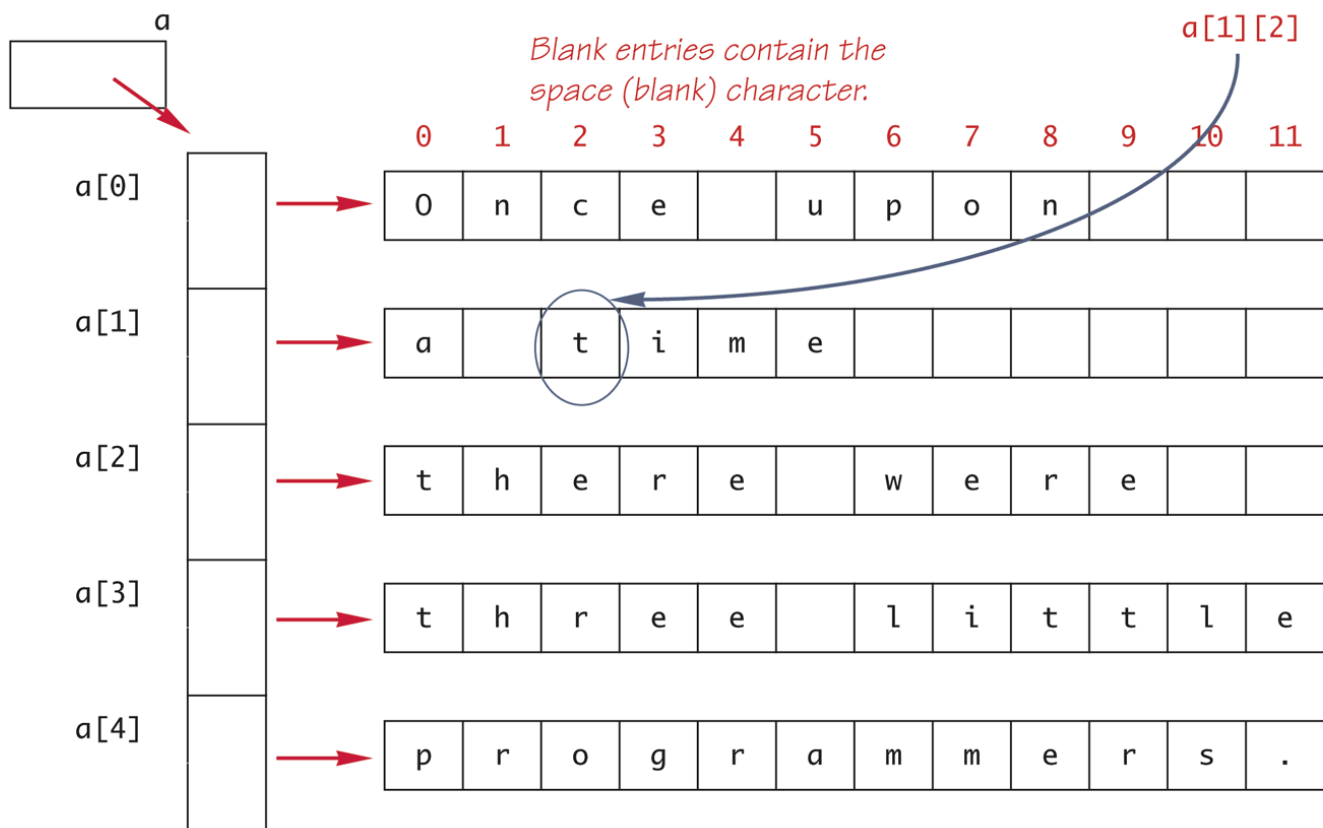
```
char[][] a = new char[5][12];
```

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).

## Display 6.17 Two-Dimensional Array as an Array of Arrays

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

*Code that fills the array is not shown.*



(continued)

```
class MultiArray {  
    public static void main (String[] args) {  
        char[][] a = new char[][] {  
            {'0', 'n', 'c', 'e', ' ', 'u', 'p', 'o', 'n', ' ', ' ', ' '},  
            {'a', ' ', 't', 'i', 'm', 'e', ' ', ' ', ' ', ' ', ' ', ' '},  
            {'t', 'h', 'e', 'r', 'e', ' ', 'w', 'e', 'r', 'e', ' ', ' '},  
            {'t', 'h', 'r', 'e', 'e', ' ', 'l', 'i', 't', 't', 'l', 'e'},  
            {'p', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'e', 'r', 's', '.'}};  
  
        for (int row = 0; row < 5; row++) {  
            for (int column = 0; column < 12; column++)  
                System.out.print(a[row][column]);  
            System.out.println('|');  
        }  
    }  
}
```

## Arrays of arrays

In Java, multidimensional arrays are really arrays of arrays.

*what is jagged array*

↓

In computer science, a jagged array, also known as a ragged array or irregular array is an array of arrays of which the member arrays can be of different lengths, producing rows of jagged edges when visualized as output.

The instance variable `length` of the multidimensional array tells us how big the first dimension is, not the total number of elements in the multidimensional array.

In the above, `a.length == 5`.

You can also ask the length of the second dimension, by asking the length of one "first level element" of the array: `a[0].length == 12`.

**Exercise:** Replace the constants 5 and 12 above by the appropriate `length` variables.

The fact that a two dimensional array is an array of arrays allows some flexibility. The line

```
double[][] = new double[3][5];
```

is equivalent to

```
double[][] a = new double[3][];  
a[0] = new double[5];  
a[1] = new double[5];  
a[2] = new double[5];
```

Note that the first line makes a the name of an array with room for 3 entries, each of which is an array of doubles that *can be of any length*.

The next three lines each create an array of doubles of size 5, but we can replace them with any sizes.

We could write

```
double[][] a = new double[3][];  
a[0] = new double[5];  
a[1] = new double[10];  
a[2] = new double[4];
```

which would make a "ragged array". It isn't a rectangle, like two-dimensional arrays in most languages. This can save a lot of memory, if the sizes are very different, but can also make things more complicated.

**Exercise:** Remove the trailing spaces in the initializer in the above code. Notice that the vertical bars at the end move, showing that the array is now ragged. Note that you must have done the above exercise correctly first.

**Exercise:** In the above code, modify the inner loop (over column) to use a for-each loop.

**Exercise:** In the above code, modify the outer loop (over row) to use a for-each loop.

## Multidimensional array parameters and return 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, just like other multidimensional variable declarations.

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

The parameter a is a two-dimensional array.

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 or variable. The following method returns a two-dimensional array of double.

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

That is, "double[][]" is just a type name like "double" or "Double".

## A grade book class

As an example of using arrays in a program, write a class `GradeBook` 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

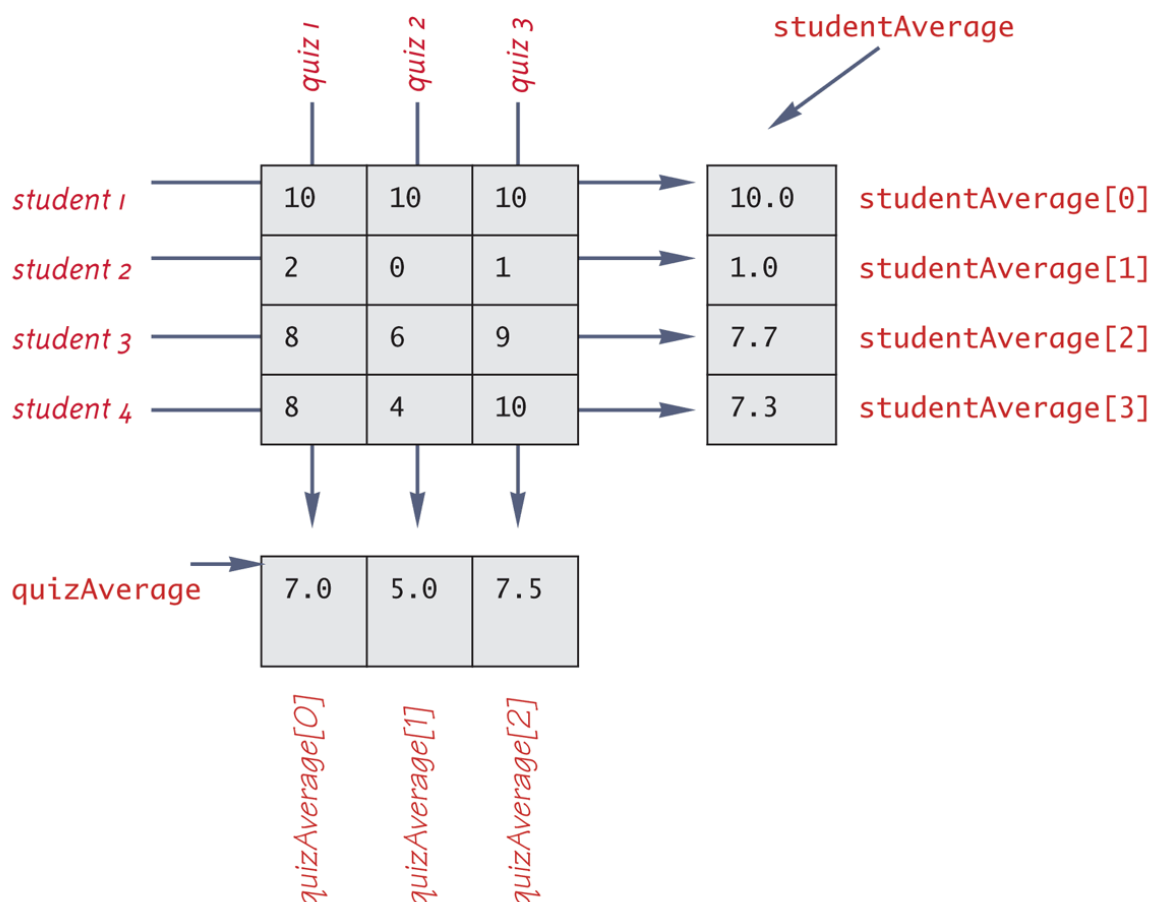
The score that student 1 received on quiz number 3 is recorded in `grade[0][2]`

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 between the three arrays.

**Display 6.19 The Two-Dimensional Array grade**



Write code to calculate the elements of `quizAverage` and `studentAverage`. You can enter the individual student grades from the keyboard, as command line arguments, or as a hard-coded

initializer:

```
{ {...}, {...}, ..., {...} }
```

Don't hard-code the size of studentAverage and quizAverage arrays; make them depend only on the input data (i.e., the grade 2D array).

You can assume that the grade array is rectangular, not ragged.



```
Grade s1Grade = new Grade();
s1Grade.setMarks(new int[]{75, 81});
s1Grade.setSubjectList(new String[]{"COMP90041", "SWEN90016"});
s1.setGrades(s1Grade);
Grade tmpGrades = s1.getGrades();
tmpGrades.setMarks(new int[]{81, 90});
s1.printStudentDetails();
```

```
[user@sahara ~]$ java GradeBook
StudentId: 1 , name: Trina
StudentId: 2 , name: Nat
StudentId: 1 , name: Trina
COMP90041 : 81
SWEN90016 : 90
```

```
class GradeBook {
    static public void main (String[] args) {
        double[][] grade = {
            {10, 10, 10},
            {2, 0, 1},
            {8, 6, 9},
            {8, 4, 10}
        };

        double[] studentAverage = new double[grade.length];
        double[] quizAverage = new double[grade[0].length];

        for (int i = 0; i < grade.length; i++) {
            double sum = 0;
            for (double sq : grade[i]) {
                sum += sq;
            }
            studentAverage[i] = sum / grade[i].length;
        }

        for (int i = 0; i < grade[0].length; i++) {
            double sum = 0;
            for (double[] student : grade)
                sum += student[i];
            quizAverage[i] = sum / grade.length;
        }

        System.out.println("Student averages:");
        for (double i : studentAverage)
            System.out.println(i);

        System.out.println("Quiz averages:");
        for (double i : quizAverage)
            System.out.println(i);
    }
}
```

```
javadoc -d docs/ Bill.java
```