

6.1: Creating and Using Arrays

Arrays - Declaration

- An Array variable is is a collection of values **of the same type** and is declared like this

```
type[] name;
```

```
boolean[] answers;  
String[] questions;  
int[] scores;  
Student[] students;
```

***Note:** Arrays in Java are Object types. As-written - these Array variables are undefined and your code will fail if you attempt to access them.*

So...

Arrays - Creation

- Arrays are created with an **initializer list** or **new**

```
boolean[] answers = {true, false, false, true};  
int[] scores = {100, 84, 95, 78};
```

```
double[] prices = new double[20];  
String[] questions = new String[5];
```

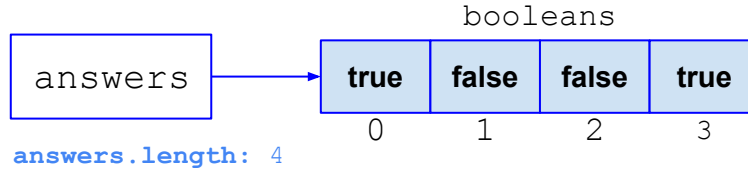
```
int numStudents = 10;  
Student[] students = new Student[numStudents];
```

Note 1: Each of these Array variables now have a value assigned to them - And can be referenced by your code.

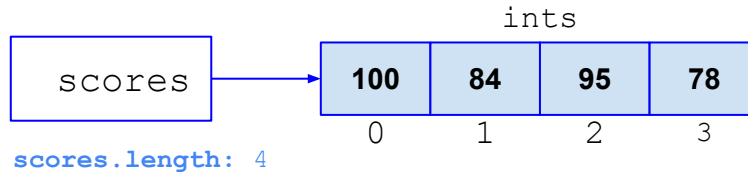
Note 2: After creation every Array has an available *length* property (which never changes)

Arrays - Creation

```
boolean[] answers = {true, false, false, true};
```

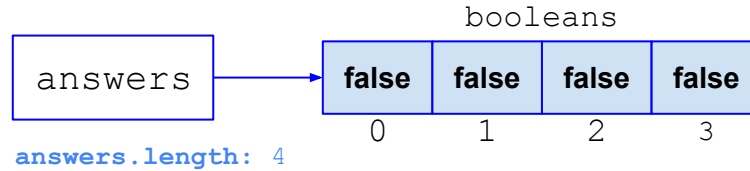


```
int[] scores = {100, 84, 95, 78};
```

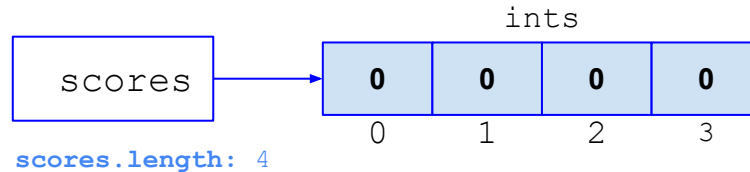


Arrays - Creation - Primitive Defaults

```
boolean[] answers = new boolean[4];
```



```
int[] scores = new int[4];
```



Arrays - Creation

Using `new` to re-assign an Array is allowed

```
boolean[] answers = {true, false, false, true};
```

This works!

```
answers = new boolean[4];
```

This also works...

```
answers = new boolean[] { true, false, false, true };
```

Arrays - Access - length

- The length of an Array can be determined via the [length property](#). **Note:** The length of a String is accessed via the [String.length\(\) method](#).

```
boolean[] answers = {true, false, false, true};  
System.out.println(answers.length);  
A> 4
```

```
String[] questions = new String[5];  
System.out.println(questions.length);  
B> 5
```

Arrays - Access - READING

- Items in an Array can be read via the `[index]` property. **Note:** Like `String` - `index` is zero-based and the range of valid `index` values is 0 to `length-1`

```
boolean[] answers = {true, false, false, true};  
System.out.print(answers[2] + ", " + answers[0]);  
C> false, true
```

```
int[] scores = {100, 84, 95, 78};  
System.out.print(scores[1] + ", " + scores[3]);  
D> 84, 78
```

Note: *Passing an out of range index will cause a `ArrayIndexOutOfBoundsException`!*

Arrays - Access - WRITING

- Items in an Array can be written via the `[index]` property. **Note:** Unlike `String` - you can change the values in an Array after it is created (however you cannot change its `length` after creation)

```
boolean[] answers = {true, false, false, true};  
answers[2] = true; answers[0] = false;  
System.out.print(answers[2] + ", " + answers[0]);  
C> true, false
```

```
int[] scores = {100, 84, 95, 78};  
scores[1] = 48; scores[3] = 87;  
System.out.print(scores[1] + ", " + scores[3]);  
D> 84, 87
```

Arrays - Access - Object Types

- Arrays that hold Object types work a little differently than those that hold primitive types
- We already saw that the length property works

```
String[] questions = new String[5];  
System.out.println(questions.length);  
B> 5
```

Arrays - Access - Object Types

- Arrays that hold Object types work a little differently than those that hold primitive types
- We already saw that the length property works

```
String[] questions = new String[5];  
System.out.println(questions.length);  
B> 5
```

- But what about reading and writing values in an Array that holds Object types?

Arrays - Access - Object Types

- But what about reading and writing values in an Array that holds Object types?

```
String[] questions = new String[5];  
System.out.println(questions[1]);  
E> null
```

```
Student[] students = new Student[5];  
System.out.println(students[1]);  
F> null
```

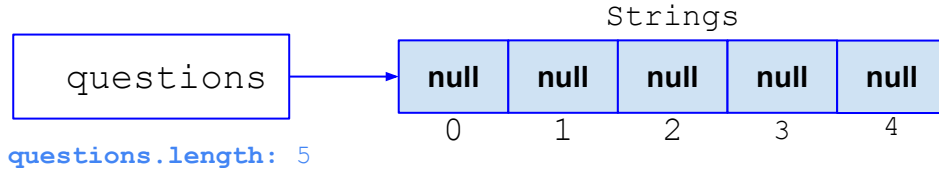
*For Arrays that hold
Object types - each slot
will be initialized to
null*

Initialize each Array slot with new

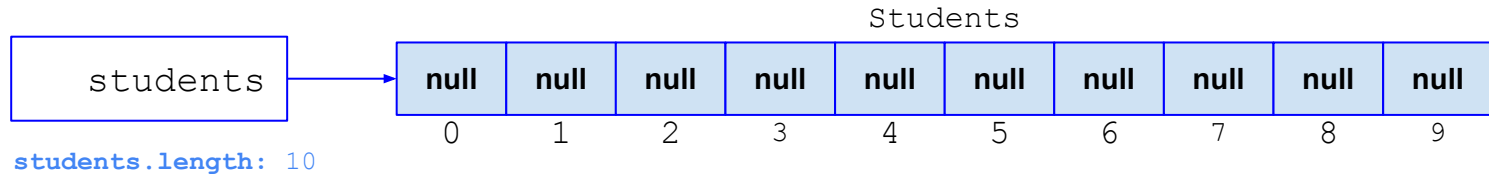
```
students[0] = new Student();  
students[1] = new Student();  
students[2] = new Student();  
...
```

Arrays - Creation

```
String[] questions = new String[5];
```

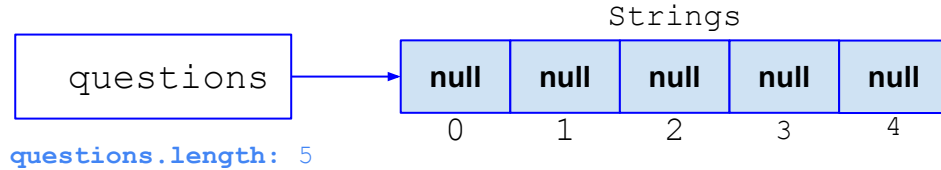


```
Student[] students = new Student[10];
```

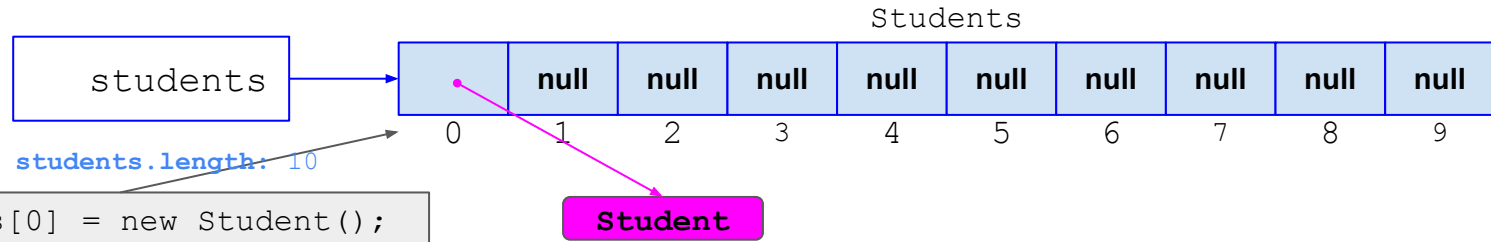


Arrays - Creation

```
String[] questions = new String[5];
```



```
Student[] students = new Student[10];
```



6.2: Traversing Arrays with `for` loops

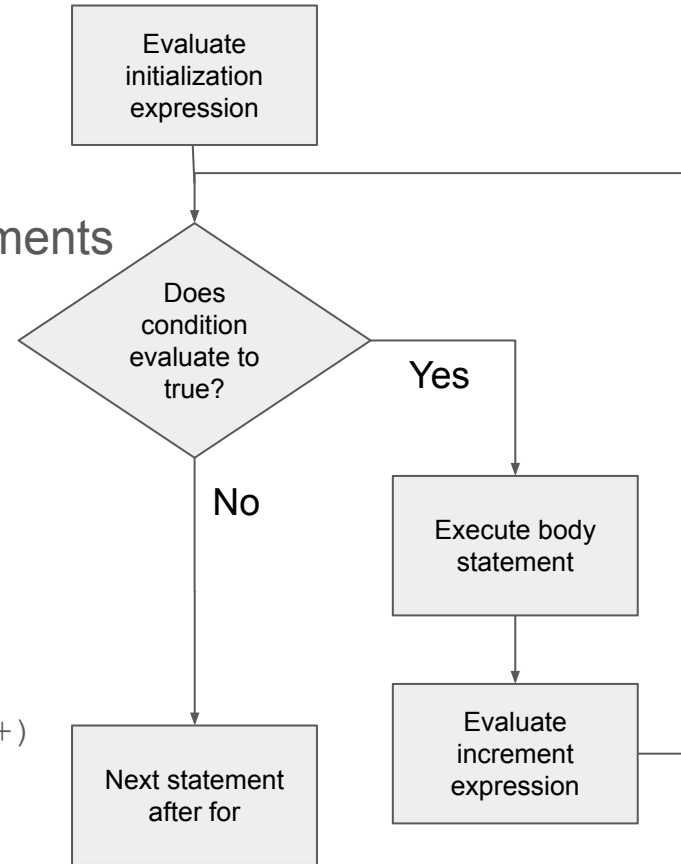
Traversing Arrays with `for` loops

- Now we can now combine this with what we have learned about accessing Arrays
- Arrays have a property called **length** and elements can be access **via [] and an index**

`for (initialization; condition; increment)`
statement

Example:

```
int[] scores = {95, 100, 91, 85 };  
for (int idx = 0; idx < scores.length; idx++)  
{  
    System.out.println(scores[idx]);  
}
```



Traversing Arrays with for loops

- Remember that the range of valid Array indexes (for non-empty Arrays) is 0 to `Array.length - 1`

```
int[] scores = {95, 100, 91, 85 };  
for (int idx = 0; idx < scores.length; idx++)  
{  
    System.out.println(scores[idx]);  
}
```

```
int[] scores = {95, 100, 91, 85 };  
for (int idx = 1; idx <= scores.length;  
idx++) {  
    System.out.println(scores[idx]);  
}
```



***This loop also
skips the first
element in the
Array!***

Note: Passing an out of range index will cause a `ArrayIndexOutOfBoundsException`!

Traversing Arrays with `for` loops

- You can use a `for` loop to traverse an Array from back to front!

```
int[] scores = {95, 100, 91, 85 };  
for (int idx = scores.length - 1; idx >= 0; idx--) {  
    System.out.println(scores[ idx]);  
}
```

Traversing Arrays with `for` loops

- You can use a `for` loop to traverse an Array from back to front!

```
int[] scores = {95, 100, 91, 85 };  
for (int idx = scores.length - 1; idx >= 0; idx--) {  
    System.out.println(scores[idx]);  
}
```

- ...or to traverse any arbitrary range of elements

```
int[] scores = {95, 100, 91, 85 };  
for (int idx = 1; idx <= 2; idx++) {  
    System.out.println(scores[idx]);  
}
```

6.3: Traversing Arrays with `for-each` loops

Traversing Arrays with for-each loops

- An alternate way to loop through Objects that support the [Iterable interface](#)

```
for (type arrayItemVariable : arrayVariable) {  
    arrayItemVariable is a copy of arrayVariable[0]  
    arrayItemVariable is a copy of arrayVariable[1]  
    arrayItemVariable is a copy of arrayVariable[...]  
    arrayItemVariable is a copy of arrayVariable[arrayVariable.length-1]  
    then the loop terminates  
}
```

Traversing Arrays with for-each loops

```
for (type arrayItemVariable : arrayVariable) {  
    arrayItemVariable resolves to arrayVariable[...]  
}
```

```
String[] colors = {"red", "orange", "purple"};
```

```
System.out.println("begin");  
for (String color: colors) {  
    System.out.println(" " + color);  
}  
System.out.println("end");
```

Output:

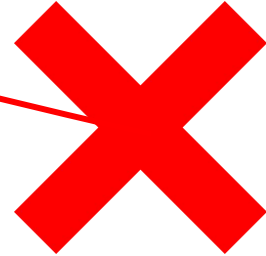
```
begin  
    red  
    orange  
    purple  
end
```

Traversing Arrays with for-each loops

- The type of the for-each variable MUST match the type of the values stored in the Array

```
String[] colors = {"red", "orange", "purple"};
```

```
for(int color: colors) {  
    System.out.println(" " + color);  
}
```



***Note:** color must be of type String since colors is an Array that contains Strings*

Comparing `for` and `for-each` loops

- `for`
 - Direct access to any element in the Array - in any order - using zero-based index and `[]`
 - You always know the index - so using parallel Arrays is easy!
 - May require more variables to efficiently operate
 - Can change the value of an Array element during the loop
- `for-each`
 - Sequential access to the elements in the Array - **must always go from first to last**
 - You do not know the index - so using Parallel Arrays is harder (impossible?)
 - May eliminate the need for extra variables (no need to use indexes to access an item)
 - Cannot change the value of an Array element during the loop

6.4: Array Algorithms

part 1

Minimum and Maximum Value

These require a "tracking value" for the smallest or largest value found so far.

25	70	9	3	15	16	19
----	----	---	---	----	----	----

minV = 25

25

9

3

3

3

3

Minimum and Maximum Value

One trick is to "seed" the tracking value with the first element, and skip it in the loop.

```
// Precondition: Array cannot be empty.
int findMinValue(int[] array) {
    int minValue = array[0];
    for (int i = 1, n = array.length; i < n; i++)
        if (array[i] < minValue) {
            minValue = array[i];
        }
    return minValue;
}
```

```
// Precondition: Array cannot be empty.
int findMaxValue(int[] array) {
    int maxValue = array[0];
    for (int i = 1, n = array.length; i < n; i++) {
        if (array[i] > maxValue) {
            maxValue = array[i];
        }
    }
    return maxValue;
}
```

Minimum and Maximum Value

With a little modification, you could return the array index instead of the value.

```
// Precondition: Array cannot be empty.
int indexOfMinValue(int[] array) {
    int minIndex = 0;
    for (int i = 1, n = array.length; i < n; i++) {
        if (array[i] < array[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}
```

```
// No preconditions... I return -1 for empty arrays.
int indexOfMinValue2(int[] array) {
    if (array.length == 0) {
        return -1;
    }
    int minIndex = 0;
    for (int i = 1, n = array.length; i < n; i++) {
        if (array[i] < array[minIndex]) {
            minIndex = i;
        }
    }
    return minIndex;
}
```

Minimum and Maximum Value of Objects

You might be dealing with an array of something other than numbers.

```
Student findYoungestStudent(Student[] students) {  
    Student youngestStudent = null;  
    for (Student student : students) {  
        if (youngestStudent == null) {  
            youngestStudent = student;  
        } else if (student.getAge() < youngestStudent.getAge()) {  
            youngestStudent = student;  
        }  
    }  
    return youngestStudent;  
}
```

Sum and Average

If dealing with ints, remember to cast to double when calculating average.
(Also known as the arithmetic mean.)

```
public int sum(int[] values) {  
    int sum = 0;  
    for (int value : values) {  
        sum += value;  
    }  
    return sum;  
}  
  
public double average(int[] values) {  
    return (double)sum(values) / values.length;  
}
```

```
public double sum(double[] values) {  
    double sum = 0;  
    for (double value : values) {  
        sum += value;  
    }  
    return sum;  
}  
  
public double average(double[] values) {  
    return sum(values) / values.length;  
}
```

Calculations aren't always over int[] or double[]...

What if you are calculating the average age of a class of Students?

```
class Student {  
    private String name;  
    private int age;  
    public Student(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public String getName() { return name; }  
    public int getAge() { return age; }  
}
```

```
private Student[] students = {  
    new Student("Alice", 16),  
    new Student("Bob", 15),  
    new Student("Carleton", 17),  
    new Student("David", 17)  
};
```

```
public double averageStudentAge(Student[] students) {  
    double sum = 0;  
    for (Student student : students) {  
        sum += student.getAge();  
    }  
    return sum / students.length;  
}
```

Transforming arrays to do calculations

It can make sense to transform an Array into another Array, and then do a calculation.

```
public double[] getStudentAges(Student[] students) {  
    int count = students.length;  
    double[] ages = new double[count];  
    for (int i=0; i<count; i++) {  
        ages[i] = students[i].getAge();  
    }  
    return ages;  
}  
  
public double averageStudentAge(Student[] students) {  
    double sum = 0;  
    for (Student student : students) {  
        sum += student.getAge();  
    }  
    return sum / students.length;  
}  
  
public double averageStudentAge2(Student[] students) {  
    return average(getStudentAges(students));  
}
```

Creating the ages Array takes time and memory... but it can make sense, depending on the situation.

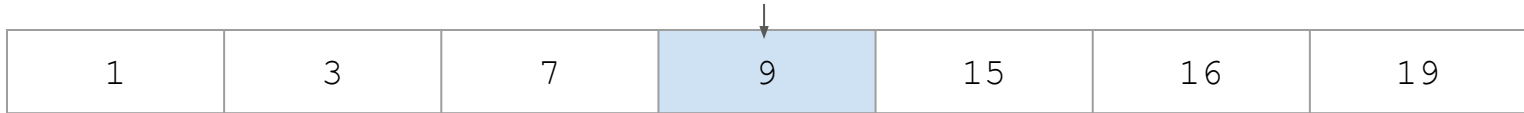
- The age data might be used more than once
- You might be interfacing with code, such as a third-party library, that doesn't know about Students
- Your math-heavy code stays in its "domain" ... it only needs to know about math, not Students, and can be reused for things other than Students.
- Performance and memory usage may not be critical

Median

Median is defined as the "middle element" of an array.

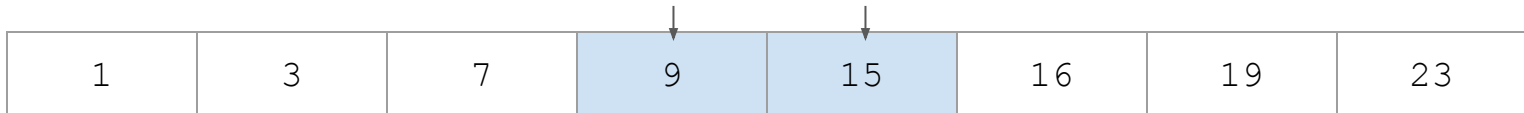
The array needs to be sorted for it to work.

If the array has odd length, the element in the middle is returned.



1	3	7	9	15	16	19
---	---	---	---	----	----	----

If the array is of even length, there isn't a "middle" ... so return the mathematical average of the two elements in the middle.



1	3	7	9	15	16	19	23
---	---	---	---	----	----	----	----

Median

The array needs to be sorted. We could declare a precondition!

We also declare a precondition that the input array must not be empty.

```
// Precondition: Array "values" must not be empty and must be sorted
// in ascending order.
public double medianOfSortedArray(double[] sortedValues) {
    int length = sortedValues.length;
    int middle = length / 2;
    if (length % 2 == 0) {
        return (sortedValues[middle-1] + sortedValues[middle]) / 2;
    } else {
        return sortedValues[middle];
    }
}
```

Median Student Age

Suppose we have an array of Students again. How do we calculate the median age?

We're able to use the tools we already built to get the student ages, and calculate the median of an array of numbers.

This is the power of **composing** methods together, and having general-purpose methods that can be applied in multiple situations. Breaking big problems into smaller problems is what CS is all about.

```
public double medianStudentAge(Student[] students) {  
    return median(getStudentAges(students));  
}
```

Mode

The **mode** of an array is the value that occurs most frequently.

17	17	9	9	9	9	1	17	19	3	3	5
----	----	---	---	---	---	---	----	----	---	---	---

What is the mode of this array?

Mode

We can calculate the mode using a nested loop. For each element, count the number of occurrences of that element, and track which element has the maximum number of occurrences.

17	17	9	9	9	9	1	17	19	3	3	5
----	----	---	---	---	---	---	----	----	---	---	---

frequency (of value at current index)

3	skip	4	skip	skip	skip	1	3	1	2	skip	1
---	------	---	------	------	------	---	---	---	---	------	---

modeValue

[illegible]

Mode

```
// Precondition: Array "values" must not be empty.
public double mode(double[] values) {
    double modeValue = values[0];
    int modeFrequency = 1;
    for (int i=1, n=values.length; i<n; i++) {
        double value = values[i];
        if (value != modeValue) {
            int frequency = 0;
            for (int j=0; j<n; j++) {
                if (values[j] == value) {
                    frequency++;
                }
            }
            if (frequency > modeFrequency) {
                modeFrequency = frequency;
                modeValue = value;
            }
        }
    }
    return modeValue;
}
```

The running time of this algorithm is $O(N^2)$.

Good? Bad?

Mode

What if the array is sorted? Does that make it easier to calculate the mode?

1	1	2	2	2	3	3	3	3	4	4	5
---	---	---	---	---	---	---	---	---	---	---	---

Mode of sorted array

With a sorted array, all of the repeated elements are adjacent to each other.

1	1	2	2	2	3	3	3	3	4	4	5
---	---	---	---	---	---	---	---	---	---	---	---

We can avoid a nested loop.


We just need to loop once, and count how many occurrences there are of each element.

We still need to track the maximum frequency, and the value associated with it.

Mode of sorted array

This algorithm is a form of **access all consecutive array elements**.

We compare each element to the previous one to see how long the "runs" are.



1	1	2	2	2	3	3	3	3	4	4	5
---	---	---	---	---	---	---	---	---	---	---	---

frequency

1	2	1	2	3	1	2	3	4	1	2	1
---	---	---	---	---	---	---	---	---	---	---	---

modeValue

1	1	1	1	2	2	2	2	3	3	3	3
---	---	---	---	---	---	---	---	---	---	---	---

modeFrequency

1	2	2	2	3	3	3	3	4	4	4	4
---	---	---	---	---	---	---	---	---	---	---	---

Mode of sorted array

```
// Precondition: Array "values" be sorted and non-empty.
public double modeOfSortedArray(double[] values) {
    double modeValue = values[0];
    int frequency = 1, modeFrequency = 1;
    for (int i=1, n=values.length; i<n; i++) {
        boolean sameAsLast = values[i-1] == values[i];
        if (sameAsLast) {
            frequency++;
        }
        if (frequency > modeFrequency) {
            modeFrequency = frequency;
            modeValue = values[i-1];
        }
        if (!sameAsLast) {
            frequency = 1;
        }
    }
    return modeValue;
}
```

Note how like calculating the minimum or maximum value, we can:

- Seed the tracking variables with the first element in the array
- And then, skip that element when iterating by starting the loop at $i=1$.

- The running time of this algorithm is $O(N)$.
- But... the array must be sorted, and sorting takes $O(N \cdot \log(N))$.

6.4: Array Algorithms

part 2

Determine number of elements meeting specific criteria

```
public int countOccurrences(double[] values, double searchValue) {  
    int count = 0;  
    for (double value : values) {  
        if (value == searchValue) {  
            count++;  
        }  
    }  
    return count;  
}
```

Search for a particular element in the array

This is known as **linear search**, the simplest (and least efficient) of search algorithms. We'll learn others later.

```
public Student findStudentByName(String name) {  
    for (Student student : students) {  
        if (student.getName().equals(name)) {  
            return student;  
        }  
    }  
    return null;  
}
```

Filter an array for all matching elements

What if you want to find all matching elements? One way is to **filter** the array into a new array.

```
public int countStudentsWithLastName(Student[] students, String lastName) {  
    int count = 0;  
    for (Student student : students) {  
        if (student.getLastName().equals(lastName)) {  
            count++;  
        }  
    }  
    return count;  
}  
  
public Student[] getStudentsWithLastName(Student[] students, String lastName) {  
    Student[] result = new Student[countStudentsWithLastName(students, lastName)];  
    int count = 0;  
    for (Student student : students) {  
        if (student.getLastName().equals(lastName)) {  
            result[count++] = student;  
        }  
    }  
    return result;  
}
```

When returning an array that is a subset of the original array, you have to decide how big to make the result array.

Here, we do it by doing another pass through the array just to count how big the result array should be, using a helper method.

Another option would be the "growable array" pattern... In Unit 7, we'll cover ArrayList which could be used for this purpose.

Search for all matching elements

Another way to find all matching elements is to build your own indexOf with startIndex parameter.

```
public int indexOfStudentWithLastName(Student[] students, String lastName, int startIndex) {  
    for (int i=startIndex, n=students.length; i<n; i++) {  
        if (students[i].getLastName().equals(lastName)) {  
            return i;  
        }  
    }  
    return -1;  
}  
  
int index = 0;  
while ((index = indexOfStudentWithLastName(students, "Smith", index)) != -1) {  
    System.out.println(students[index]);  
    index++;  
}
```

Determine if at least one element has a particular property

The output of this type of algorithm is a boolean. As soon as you find the first element with the desired property, you can return true.

```
boolean isAnyoneYoungerThan(Student[] students, int age) {  
    for (Student student : students) {  
        if (student.getAge() < age) {  
            return true;  
        }  
    }  
    return false;  
}
```


Determine if all elements have a particular property

This is essentially the same, except we're trying to ensure that ALL elements have some property. As soon as we find an element that doesn't, we return false.

```
// Precondition: Students must be a non-empty array.
boolean isEveryoneAgeOrOlder(Student[] students, int minAge) {
    for (Student student : students) {
        if (student.getAge() < minAge) {
            return false;
        }
    }
    return true;
}
```

Universal / Existential Quantifiers

If all of the numbers are even, then there are no odd numbers.

If there are any odd numbers, then not all of the numbers are even.

```
// Precondition: Array "values" must not be empty
public boolean allEven(int[] values) {
    for (int value : values) {
        if (value % 2 != 0) {
            return false;
        }
    }
    return true;
}
```

```
// Precondition: Array "values" must not be empty
public boolean anyOdd(int[] values) {
    for (int value : values) {
        if (value % 2 != 0) {
            return true;
        }
    }
    return false;
}

// Precondition: Array "values" must not be empty
public boolean anyOdd2(int[] values) {
    return !allEven(values);
}
```

Reverse an array (in place)

```
public void reverseInPlace(int[] values) {  
    for (int i=0, n=values.length; i<n/2; i++) {  
        int temp = values[i];  
        values[i] = values[n-i-1];  
        values[n-i-1] = temp;  
    }  
}
```

```
public void reverseInPlace2(int[] values) {  
    for (int i=0, j=values.length-1; i<j; i++, j--) {  
        int temp = values[i];  
        values[i] = values[j];  
        values[j] = temp;  
    }  
}
```

The top implementation is fine, but the bottom one uses *i* and *j* instead of just *i*.

I find it easier to reason about what this algorithm is doing by having two index counters, "racing" toward each other from each end of the array.

Computers have many **registers** for storage of frequently used variables, so there is really no additional cost to having two variables instead of one. It can be even faster, since less arithmetic is performed.

Return a reversed copy of an array

```
public int[] reversedCopy(int[] values) {  
    int n = values.length;  
    int[] result = new int[n];  
    for (int i=0; i<n; i++) {  
        result[i] = values[n-i-1];  
    }  
    return result;  
}
```

```
public int[] reversedCopy2(int[] values) {  
    int n = values.length;  
    int[] result = new int[n];  
    for (int i=0, j=n-1; i<n; i++, j--) {  
        result[i] = values[j];  
    }  
    return result;  
}
```

```
public int[] reversedCopy3(int[] values) {  
    int[] result = Arrays.copyOf(values, values.length);  
    reverseInPlace(result);  
    return result;  
}
```

If you're returning a copy of an array in reverse order, you don't have to do any swapping.

It could still be helpful to use two counters instead of one.

You also could just not write the code at all... and leverage the reverse-in-place algorithm we just wrote. It will take a little more CPU time, though, since the array will first be copied, then reversed.

Check for presence of duplicate elements

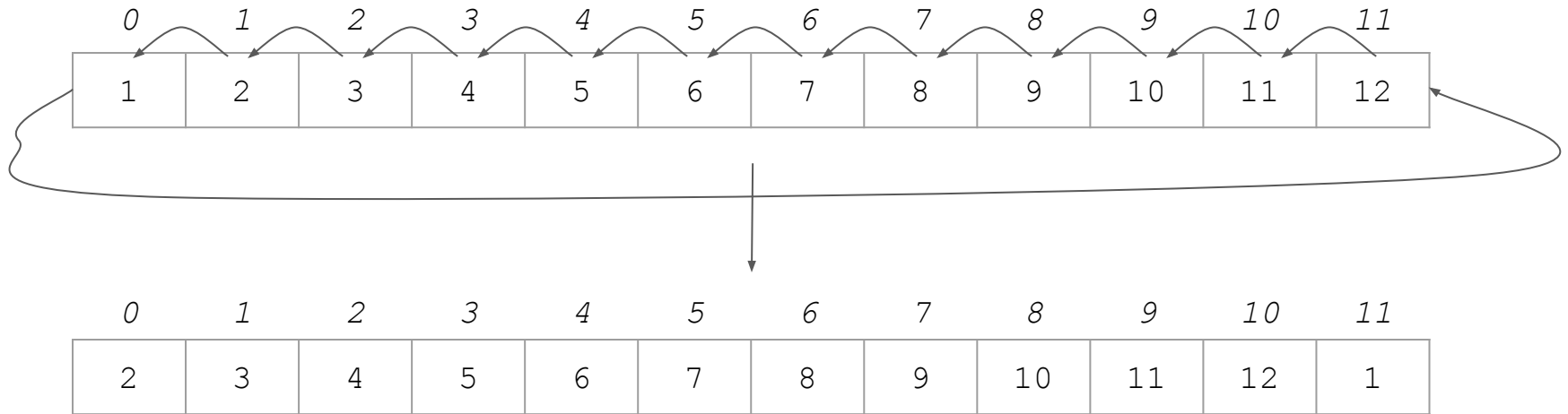
```
public boolean hasDuplicates(double[] values) {  
    for (int i=0, n=values.length; i<n; i++) {  
        for (int j=i+1; j<n; j++) {  
            if (values[i] == values[j]) {  
                return true;  
            }  
        }  
    }  
    return false;  
}
```

```
// Precondition: "values" must be sorted  
public boolean sortedArrayHasDuplicates(double[] values) {  
    for (int i=1, n=values.length; i<n; i++) {  
        if (values[i-1] == values[i]) {  
            return true;  
        }  
    }  
    return false;  
}
```

Shift or rotate an array

Here, we rotate an array of numbers to the left by 1 position.

$a[i] = a[i+1]$ for all i . The first element gets moved to the last position.



Rotate array to the left, in place

```
// Precondition: Array "values" must not be empty.  
public void rotateLeft(double[] values) {  
    double firstValue = values[0];  
    for (int i=0, n=values.length; i<n-1; i++) {  
        values[i] = values[i+1];  
    }  
    values[values.length-1] = firstValue;  
}
```

0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

0	1	2	3	4	5	6	7	8	9	10	11
2	3	4	5	6	7	8	9	10	11	12	1

An arrow points from the value 7 at index 6 in the first array to the value 7 at index 6 in the second array, illustrating the shift of elements.

Rotate array to the left multiple positions, in place

There are other ways... but this is a quick and dirty way to do it.

```
// Precondition: Array "values" must not be empty
public void rotateLeftMultiple(double[] values, int rotateAmount) {
    while (rotateAmount > 0) {
        rotateLeft(values);
        rotateAmount--;
    }
}
```


Rotate array to the right, in place

```
// Precondition: Array "values" must not be empty.
public void rotateRight(double[] values) {
    int n = values.length;
    double lastValue = values[n-1];
    for (int i=n-1; i>0; i--) {
        values[i] = values[i-1];
    }
    values[0] = lastValue;
}
```

0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

↓

0	1	2	3	4	5	6	7	8	9	10	11
12	1	2	3	4	5	6	7	8	9	10	11

Remove element at specific index from an array

This is essentially like rotating left, but starting at a particular spot. Here, we fill in the end with **null**.

```
public void removeStudentAt(Student[] students, int targetIndex) {  
    int n = students.length;  
    for (int i=targetIndex; i<n; i++) {  
        students[i] = students[i+1];  
    }  
    students[n-1] = null;  
}
```

Insert element at specific index in array

This is very similar to rotating the array right, but starting at a specific index and not "wrapping."

```
public void insertStudentAt(Student[] students, Student newStudent, int targetIndex) {  
    int n = students.length;  
    for (int i=n-1; i>targetIndex; i--) {  
        students[i] = students[i-1];  
    }  
    students[targetIndex] = newStudent;  
}
```

The last element will be lost, so you'd have to make sure there is empty space at the end!