2023-04-19

Upcoming Schedule

Monday	Wednesday	Friday
04/17/2023 (90) • Review: Unit 5, Unit 9 **• AP CS Question 2: Classes**	**04/19/2023 (90)** • Review: Units 6-7 **• AP CS Question 3: Array/ArrayList**	**04/21/2023 (45)** • Review: Unit 10 • More recursion exercises like we did on Apr-7
04/24/2023 (90) • Review: Unit 8 **• AP CS Question 4: 2D Array**	**04/26/2023 (90)** **• AP CS Multiple Choice Game**	**04/28/2023 (45)** • Review: Unit 7, Unit 10 • Algorithms: Iterative/recursive binary search, selection sort, insertion sort, merge sort
05/01/2023 **• FINAL**	**05/03/2023** **• AP EXAM**	

Units 6-7
AP CS FRQ 3

(Array and ArrayList)

6.1: Creating and Using Arrays

Arrays - Declaration

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

```
type[] name;
```

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

Note: *type* *name*[] is also valid syntax for creating an array.

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

So...

Arrays - Creation

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

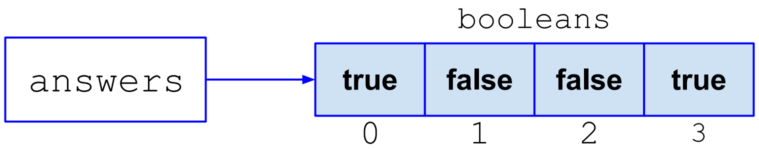
```
boolean[] answers = {true, false, false, true};
int[] scores = {100, 84, 95, 78};

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};
```



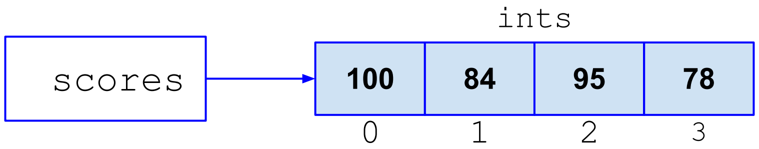
answers

booleans

true	false	false	true
0	1	2	3

`answers.length: 4`

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



scores

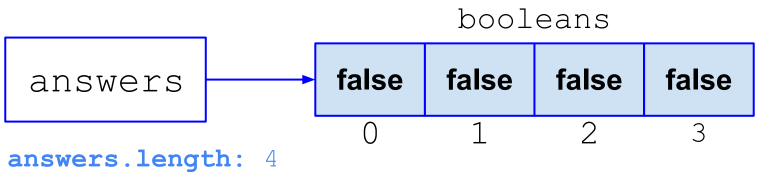
ints

100	84	95	78
0	1	2	3

`scores.length: 4`

Arrays - Creation - Primitive Defaults

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



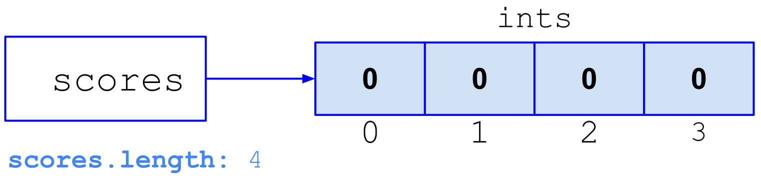
answers

booleans

false	false	false	false
0	1	2	3

answers.length: 4

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



scores

ints

0	0	0	0
0	1	2	3

scores.length: 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 using the `[]` operator, e.g. `array[index]`
  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 assigned with via the `[]` operator.
  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 properly 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 properly works

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

But what about reading and writing values in an Array that hold 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 Stude
System.out.println(students[1]
F> null
```

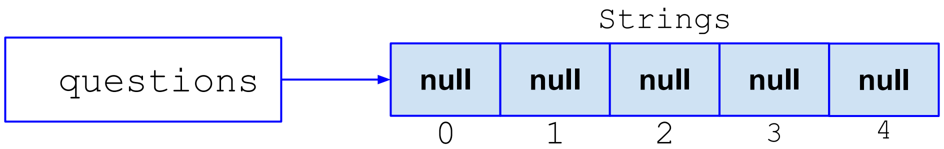
For Arrays that hold Object types - each slot will be be initialized to `null`

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];
```



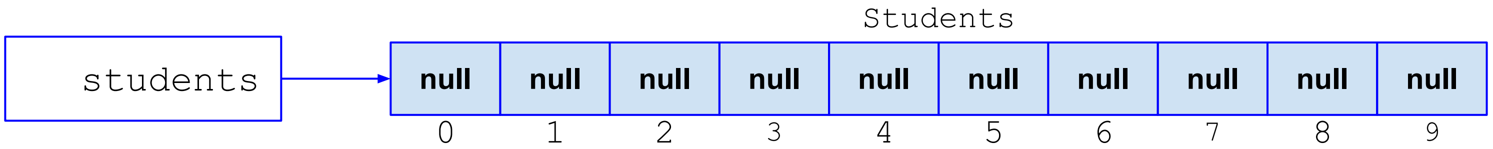
Strings

questions

null	null	null	null	null
0	1	2	3	4

questions.length: 5

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



Students

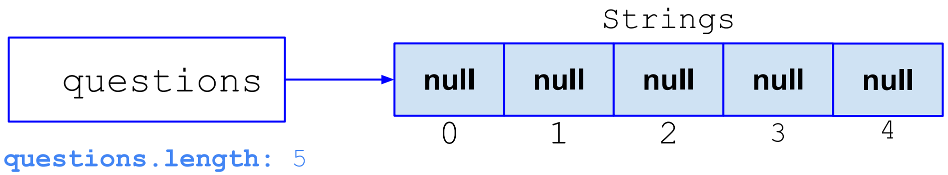
students

null	null	null	null	null	null	null	null	null	null
0	1	2	3	4	5	6	7	8	9

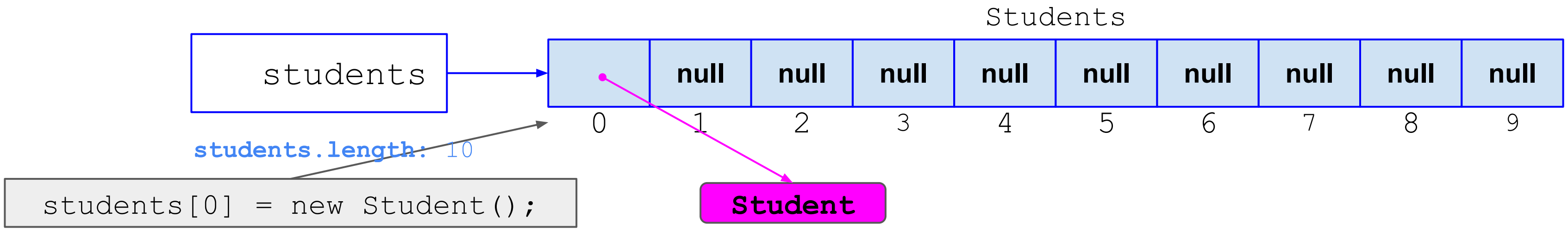
students.length: 10

Arrays - Creation

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



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



```
students[0] = new Student();
```

6.2: Traversing Arrays with `for` loops

Traversing Arrays with `for` loops

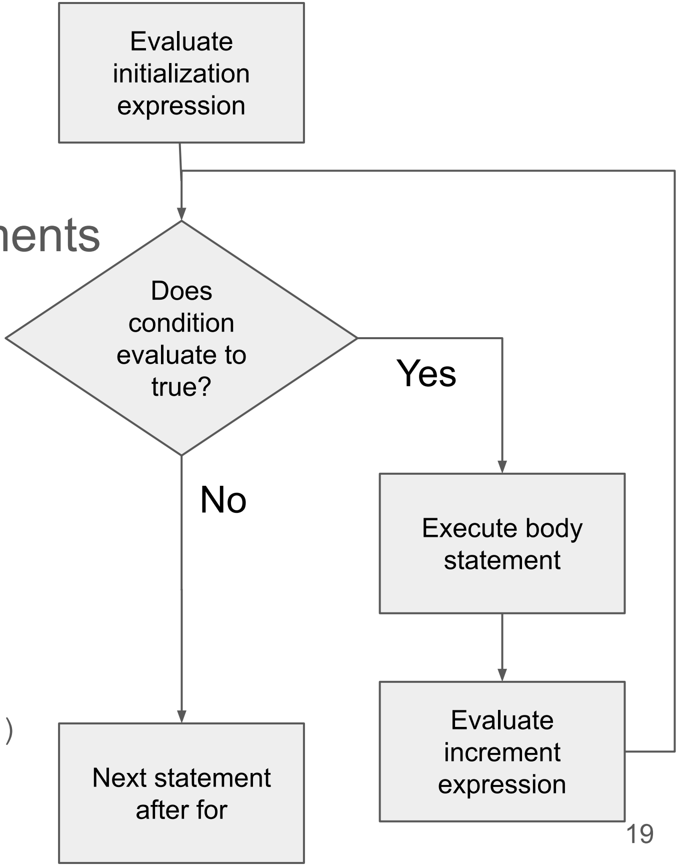
- Now we can now combine this with what we have learned about accessing Arrays
- Arrays have a property called **`length`** and elements can be access **via `[]` and an `index`**

`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]);
}
```


Evaluate initialization expression
Does condition evaluate to true?
Yes
No
Execute body statement
Evaluate increment expression
Next statement after for

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

7.1: `ArrayList`

ArrayList

- ArrayLists are collections of values of the same Object type; But have different declaration syntax than Arrays; **Primitive types (int, boolean, double, etc.) are not supported**

```
ArrayList<type> name;
```

- Examples

```
~~ArrayList<boolean> answers;~~ ** PRIMITIVE TYPES UNSUPPORTED **
ArrayList<Boolean> answers;
~~ArrayList<int> scores;~~ ** PRIMITIVE TYPES UNSUPPORTED **
ArrayList<Integer> scores;
ArrayList<String> questions;
ArrayList<Student> students;
```

- **Important:** You must import ArrayList prior to using it

```
import java.util.ArrayList;
```

Generics / Generic Types

- `ArrayList` is an example of a class that uses a Generic Type

```
ArrayList<type> name;
```

- Generic Types are an option when the **same code** can be used across a variety of data types – and frees you from needing to create an overloaded method for every type
- `ArrayList` is able to use Generic Types because the internals assume everything is a `Object` type (and all `Object` types share the functionality required for `ArrayList` to work)
- You can read more about Generics in the online Java documentation
  - Oracle Java Documentation: Why Use Generics?

ArrayList

- Like Arrays, you must initialize `ArrayLists` prior to using them; The most common usage is with the no-parameter Constructor

```
ArrayList<Boolean> answers = new ArrayList<Boolean>();
ArrayList<Integer> scores = new ArrayList<Integer>();
ArrayList<String> questions = new ArrayList<String>();
ArrayList<Student> students = new ArrayList<Student>();
```

- **Note:** There are two other `ArrayList` Constructors that you can explore on your own

```
ArrayList<type> name = new ArrayList<type>(Collection<type> c);
ArrayList<type> name = new ArrayList<type>(init initialCapacity);
```

`ArrayList`

- Unlike Arrays, `ArrayLists` automatically manage their memory usage as you `ArrayList.add()` and `ArrayList.remove()` elements to/from the the `ArrayList`
- Unlike Arrays, `ArrayLists` do not have a `length` property that indicates the fixed-size of the Array; They have the `ArrayList.size()` method that indicates the current number of elements included in the `ArrayList`
- `ArrayLists` have an internal capacity - which you cannot access - that grows and shrinks as needed to ensure elements can be quickly added. **The default capacity is 10.**
- The capacity is adjusted to ensure that the there is enough free space to quickly accommodate new items via `ArrayList.add()`; But not so much excess free space that available memory is wasted

Array vs `ArrayList`

Array

true	false	true	*false*	*false*	*false*	*false*	*false*	*false*	*false*
0	1	2	3	4	5	6	7	8	9

```
boolean[] answers = new boolean[10];
answers[0] = true; answers[1] = false; answers[2] = true;
answers.length == 10
answers[3-9] are set to default values
```

ArrayList

true	false	true							
0	1	2	3	4	5	6	7	8	9

```
ArrayList<Boolean> answers = new ArrayList<Boolean>();
answers.add(true); answers.add(false); answers.add(true);
answers.size() == 3
answers[3-9] are unused pre-allocated capacity
```

`ArrayList` Methods

`ArrayList` Methods

- **`add()`**
- `clear()`
- `get()`
- `isEmpty()`
- `remove()`
- `removeRange()`
- `set()`
- `size()`

`ArrayList` `index` values are zero-based (just like Arrays)

Signatures

- `boolean add(E obj)`
- `void add(int index, E obj)`

Overview

- Add an item either to the end of the `ArrayList` (**always returns `true`**) or at the specified `index` (existing items will shift right; their index values will increase by 1)
  - The first version of **`add()`** always returns `true` because `ArrayList` implements the `Collection` interface - which **can** be implemented by other classes to restrict the creation of duplicate or `null` elements (`ArrayList` has no such restrictions)
- Automatically increases the `ArrayList` capacity as needed
- Will throw IndexOutOfBoundsException if `index` is out of range `(index < 0 || index > size())`

`ArrayList` Methods

- `add()`
- **`clear()`**
- `get()`
- `isEmpty()`
- `remove()`
- `removeRange()`
- `set()`
- `size()`

Signatures

- `void clear()`

Overview

- Removes all elements from the `ArrayList`
- After this call `ArrayList.size() == 0`
- Automatically decreases the `ArrayList` capacity as needed

`ArrayList` Methods

- `add()`
- `clear()`
- **`get()`**
- `isEmpty()`
- `remove()`
- `removeRange()`
- `set()`
- `size()`

`ArrayList` `index` values are zero-based (just like Arrays)

Signatures

- `E get(int index)`

Overview

- Returns the element at the specified position in the `ArrayList`
- You must use this method to access the items in an `ArrayList`; `ArrayList` does not support the `[]` syntax of Arrays
- Will throw IndexOutOfBoundsException if `index` is out of range `(index < 0 || index >= size())`

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- **`isEmpty()`**
- `remove()`
- `removeRange()`
- `set()`
- `size()`

Signatures

- `boolean isEmpty()`

Overview

- Returns true if the `ArrayList` has no items

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- `isEmpty()`
- **`remove()`**
- `removeRange()`
- `set()`
- `size()`

`ArrayList index` values are zero-based (just like Arrays)

It really is `Object`, not `E`, here. Historical reasons.

Signatures

- `boolean remove(Object obj)`
- `E remove(int index)`

Overview

- Removes the first item from the `ArrayList` that matches `obj`; or at the specified `index` (existing items will shift left; their index values will decrease by 1)
  - `remove(Object obj)` returns `true/false` if an element in the `ArrayList` returns `true` for `obj.equals(element)` `(or obj == null == element)` and was removed
    - Note: Does **not use** Object equality (`obj == element`)
  - `remove(int index)` returns the element that was removed from the `ArrayList`
- Automatically decreases the `ArrayList` capacity as needed
- Will throw IndexOutOfBoundsException if `index` is out of range `(index < 0 || index >= size())`

ArrayList Methods

- add()
- clear()
- get()
- isEmpty()
- remove()
- **removeRange()**
- set()
- size()

ArrayList index values are zero-based (just like Arrays)

Signatures

- `void removeRange(int fromIndex, int toIndex)`

Overview

- Removes all of the elements whose index is between fromIndex (inclusive) and toIndex (exclusive). Shifts any succeeding elements to the left (reduces their index).
- Automatically decreases the ArrayList capacity as needed
- Will throw IndexOutOfBoundsException if fromIndex or toIndex is out of range (fromIndex < 0 || fromIndex >= size() || toIndex > size() || toIndex < fromIndex)

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- `isEmpty()`
- `remove()`
- `removeRange()`
- **`set()`**
- `size()`

`ArrayList` `index` values are zero-based (just like Arrays)

Signatures

- `E set(int index, E element)`

Overview

- Replaces the element at the specified position in this `ArrayList` with the specified element.
- Returns the element that was removed from the `ArrayList` at `index`
- You must use this method to access the items in an `ArrayList`; `ArrayList` does not support the `[]` syntax of Arrays
- Will throw IndexOutOfBoundsException if `index` is out of range `(index < 0 || index >= size())`

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- `isEmpty()`
- `remove()`
- `removeRange()`
- `set()`
- **`size()`**

Signatures

- `int size()`

Overview

- Returns the number of elements in this `ArrayList`

7.3: Traversing `ArrayLists` with Loops

Traversing `ArrayLists`

- `ArrayLists` support the same mechanisms you used when traversing Arrays - `while`, `for`, `for-each` - with the following differences

Operation	**`Array`**	**`ArrayList`**

length/size	`Array.length`
	`(property)`
		`ArrayList.size()`
		`(method)`

read	`value = array[index];`	`value = arrayList.get(index);`
write	`array[index] = value;`	`arrayList.set(index, value);`

Traversing `ArrayLists`

Array - for loop

```java
Integer[] array = {1, 2, 3, 4, 5};
for (int idx = 0 ; idx <  array.length ; idx++) {
  int value = array[idx];
  System.out.println(value);
  array[idx] = value + 1;
}
```

ArrayList - for loop

```java
Integer[] array = {1, 2, 3, 4, 5};
ArrayList<Integer> arrayList = new ArrayList<Integer>(Arrays.asList(array));
for (int idx = 0 ; idx <  arrayList.size() ; idx++) {
  int value = arrayList.get(idx);
  System.out.println(value);
  arrayList.set(idx, value + 1);
}
```

Traversing `ArrayLists`

`Array & ArrayList - for loop`
`ArrayIndexOutOfBoundsException`

`This exception will be thrown if you try to access the item at an index less than 0 or greater than the number of items in the Array or ArrayList`

Traversing `ArrayLists`

Array - for-each loop

```
Integer[] array = {1, 2, 3, 4, 5};
for (Integer value : array) {
  System.out.println(value);
}
```

ArrayList - for-each loop

```
Integer[] array = {1, 2, 3, 4, 5};
ArrayList<Integer> arrayList = new ArrayList<Integer>(Arrays.asList(array));
for (Integer value : arrayList) {
  System.out.println(value);
}
```

Traversing `ArrayLists`

ArrayList for-each loop **ConcurrentModificationException**
This exception will be thrown if you try to add or remove items from an ArrayList while traversing that ArrayList with a for-each loop

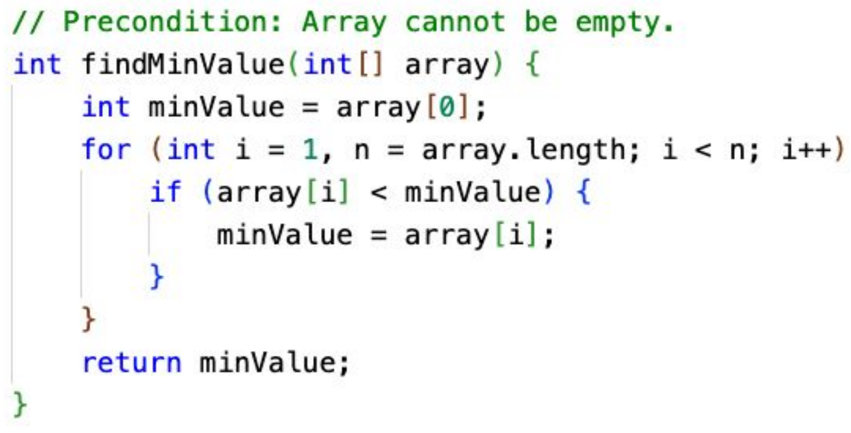
6.4: Array Algorithms

Minimum and Maximum Value

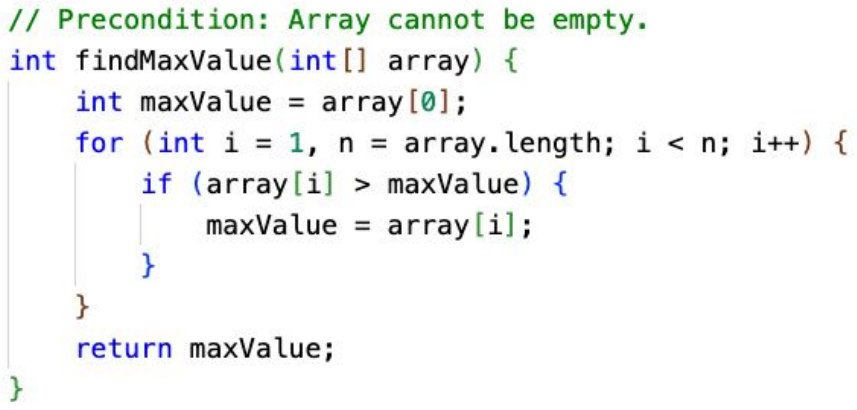
These require a "tracking value" for the smallest or largest value found so far.

	25	70	9	3	15	16	19
minV =	25	25	9	3	3	3	3

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;
}
```

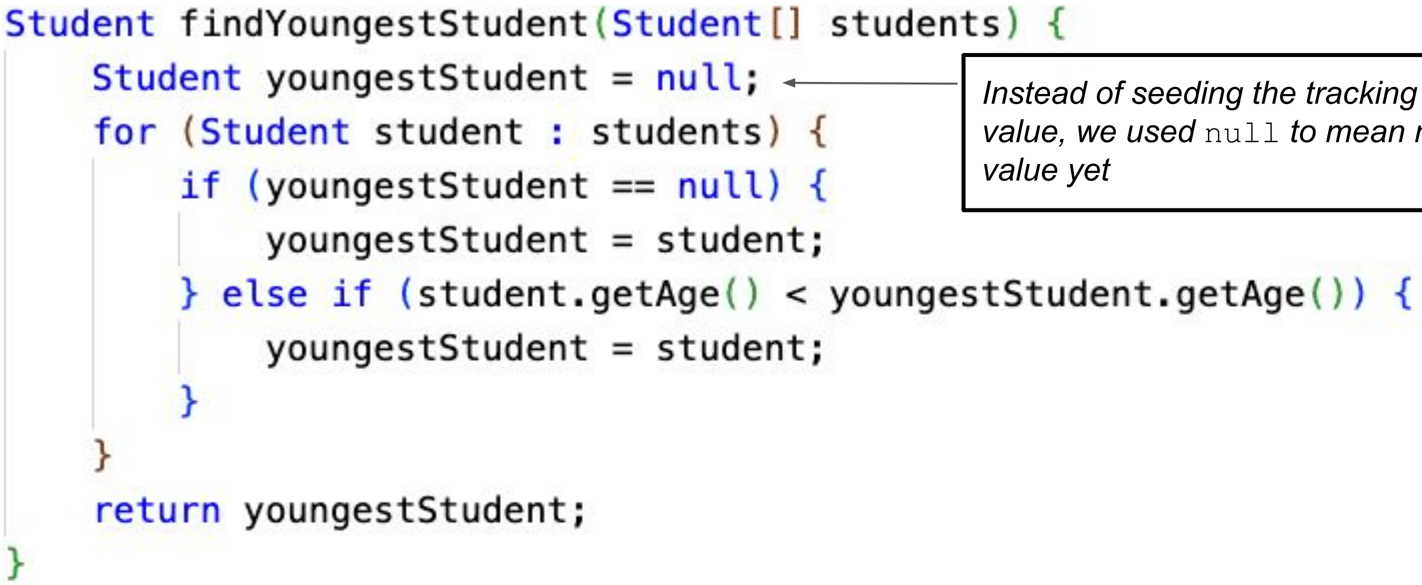


```
// Precondition: Array cannot be empty.
int findMaxValue(int[] array) {
    int maxValue = array[0];
    for (int i = 1, n = array.length; i < n; i++) {
        if (array[i] > maxValue) {
            maxValue = array[i];
        }
    }
    return maxValue;
}
```

Minimum and Maximum Value of Objects

You might be dealing with an array of something other than numbers, and the array might possibly be empty.

// Precondition: No element in students will be null, but students.length may be 0

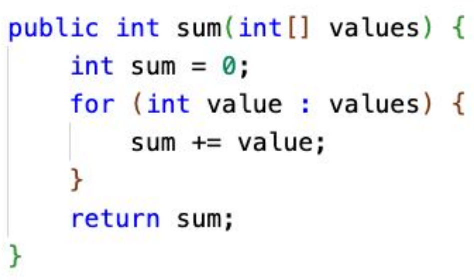


```
Student findYoungestStudent(Student[] students) {
    Student youngestStudent = null;
    for (Student student : students) {
        if (youngestStudent == null) {
            youngestStudent = student;
        } else if (student.getAge() < youngestStudent.getAge()) {
            youngestStudent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

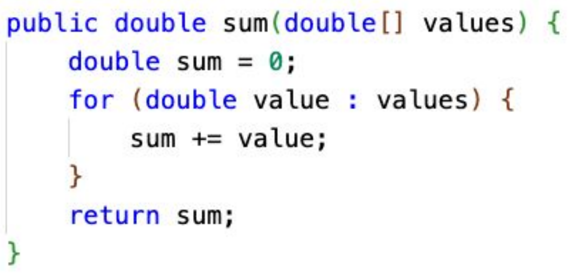
Instead of seeding the tracking value, we used `null` *to mean no value yet*

Sum and Average

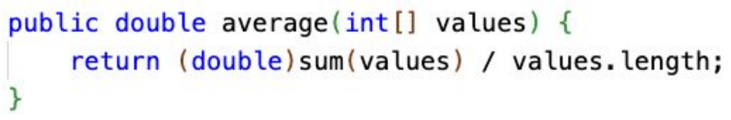
If dealing with `int`, remember to promote/cast to `double` when calculating average. (Also known as the arithmetic mean.)



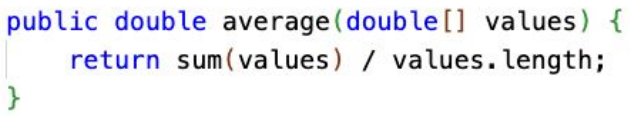
```java
public int sum(int[] values) {
    int sum = 0;
    for (int value : values) {
        sum += value;
    }
    return sum;
}
```



```java
public double sum(double[] values) {
    double sum = 0;
    for (double value : values) {
        sum += value;
    }
    return sum;
}
```



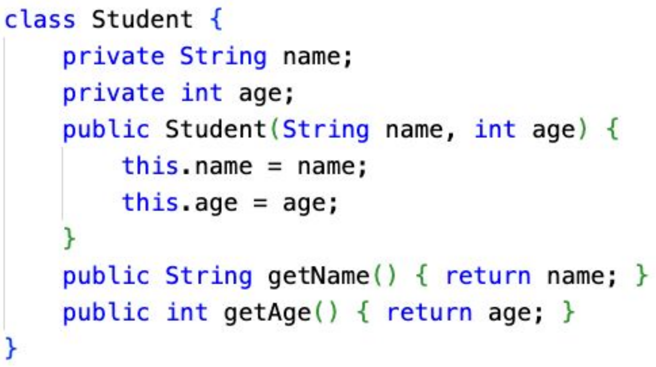
```java
public double average(int[] values) {
    return (double)sum(values) / values.length;
}
```



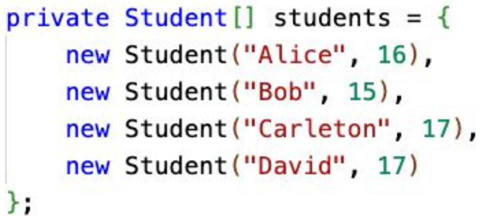
```java
public double average(double[] values) {
    return sum(values) / values.length;
}
```

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

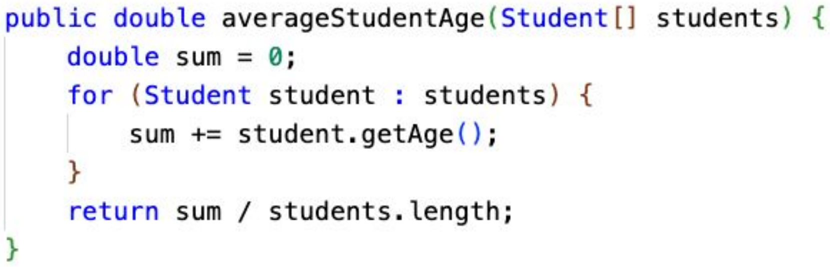
What if you are calculating the average age of a class of Students?



```java
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; }
}
```



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



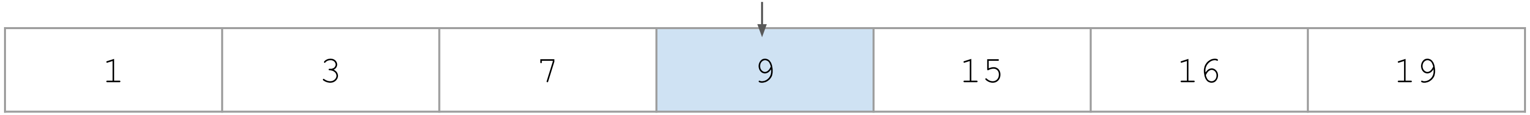
```java
public double averageStudentAge(Student[] students) {
    double sum = 0;
    for (Student student : students) {
        sum += student.getAge();
    }
    return sum / students.length;
}
```

Median

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

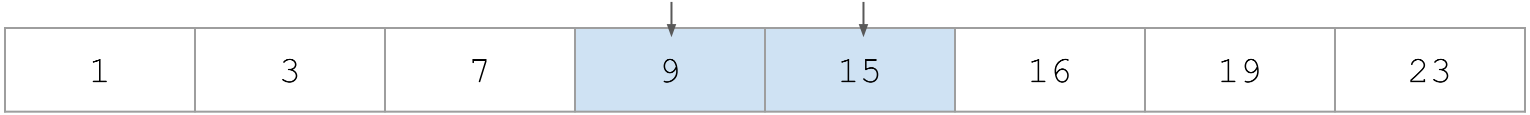
The array needs to be sorted for it to work.

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.

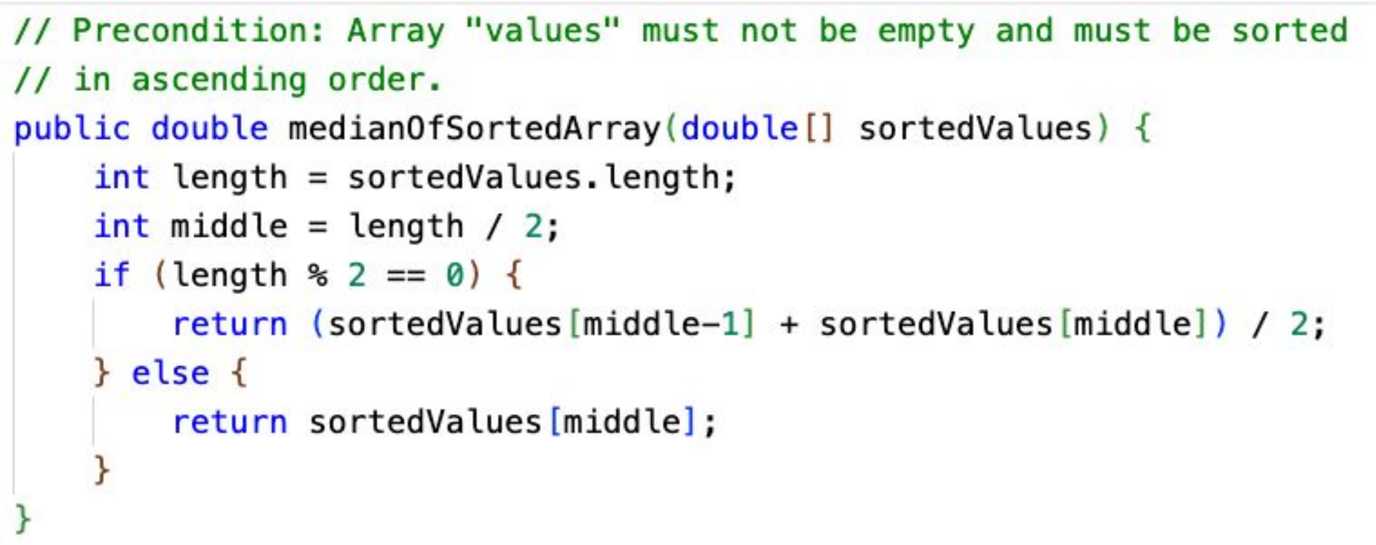


1	3	7	9	15	16	19	23

Median

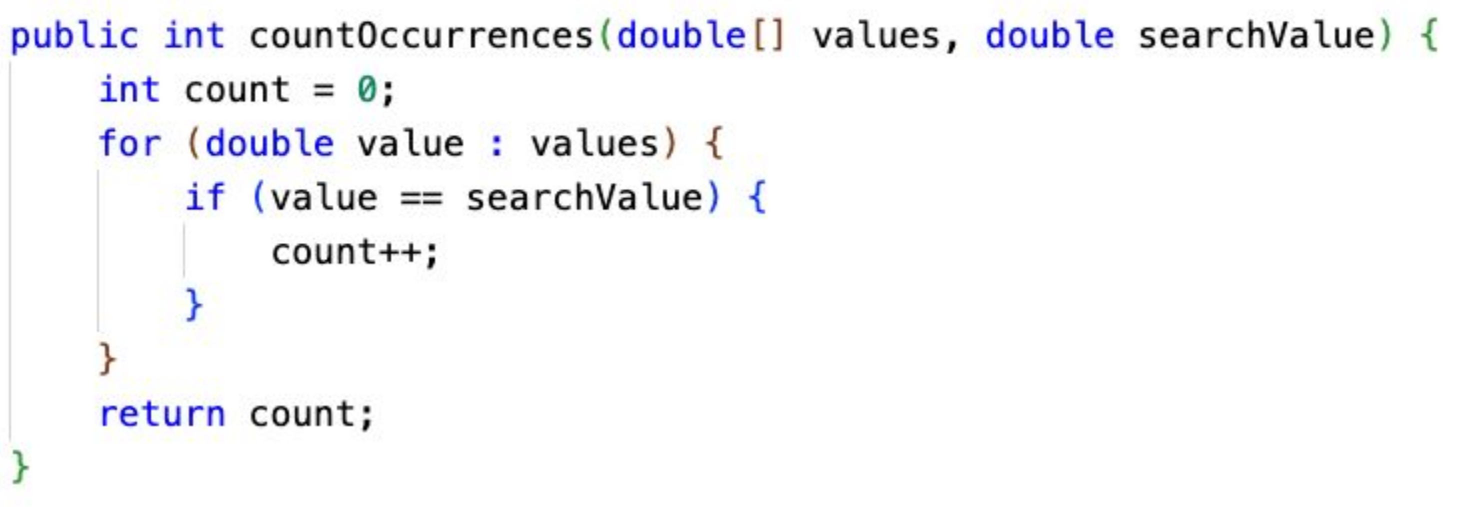
The array needs to be sorted. We could declare a precondition!

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



```java
// 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];
    }
}
```

Determine number of elements meeting specific criteria



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

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?

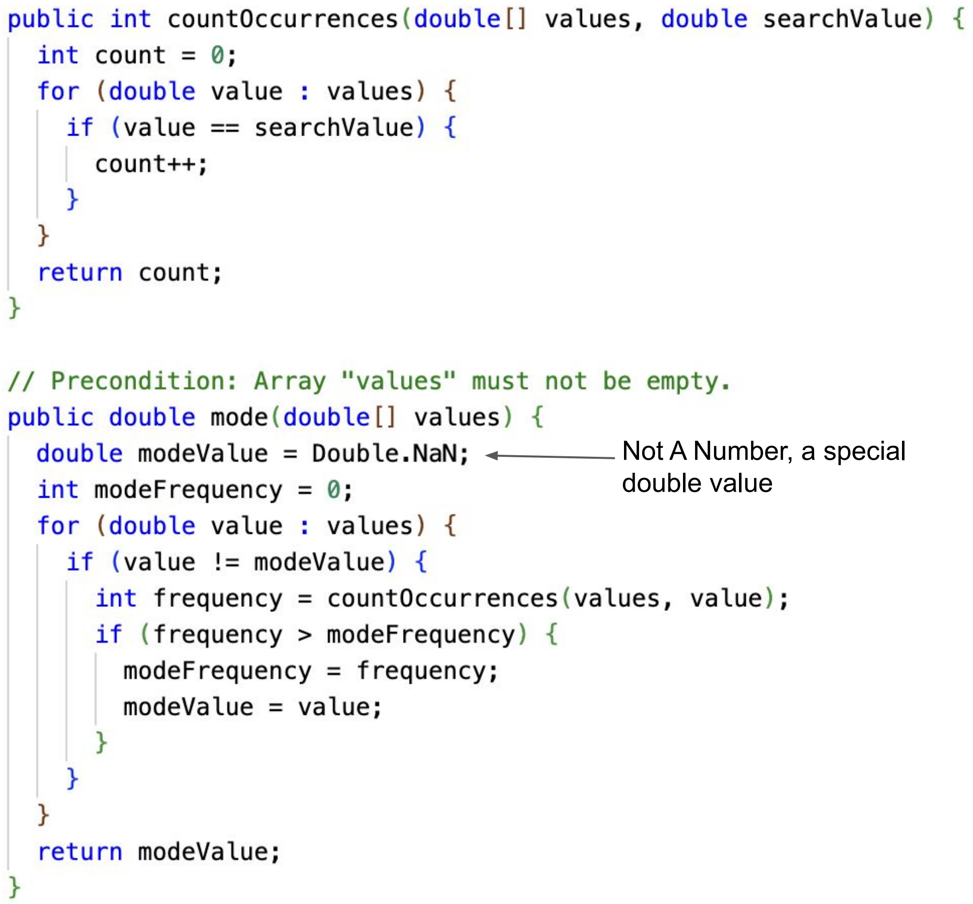
Mode

Mode could be implemented with a nested loop... an outer loop to consider each element, and and inner loop to count up occurrences of that element.

Here, we broke down the problem and used a helper method (the one from the last slide!) to count the occurrences.

The running time of this algorithm is O(N^2), with or without the helper method.

With a precondition that the array is sorted, the running time can be reduced to O(N).



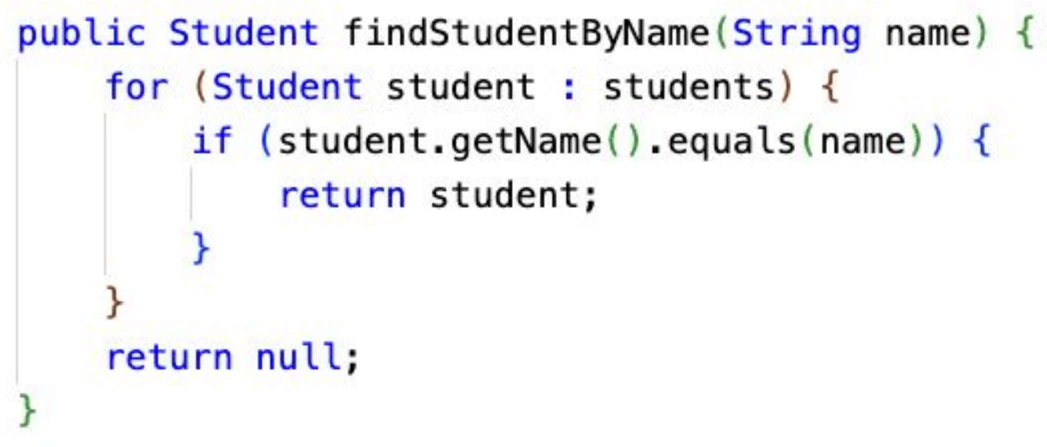
```java
public int countOccurrences(double[] values, double searchValue) {
  int count = 0;
  for (double value : values) {
    if (value == searchValue) {
      count++;
    }
  }
  return count;
}

// Precondition: Array "values" must not be empty.
public double mode(double[] values) {
  double modeValue = Double.NaN;
  int modeFrequency = 0;
  for (double value : values) {
    if (value != modeValue) {
      int frequency = countOccurrences(values, value);
      if (frequency > modeFrequency) {
        modeFrequency = frequency;
        modeValue = value;
      }
    }
  }
  return modeValue;
}
```

Not A Number, a special double value

Search for a particular element in the array

L**inear search** or **sequential search** is the simplest (and least efficient) of search algorithms, with O(N) running time. If you have an unordered array, it may be the best you can do.

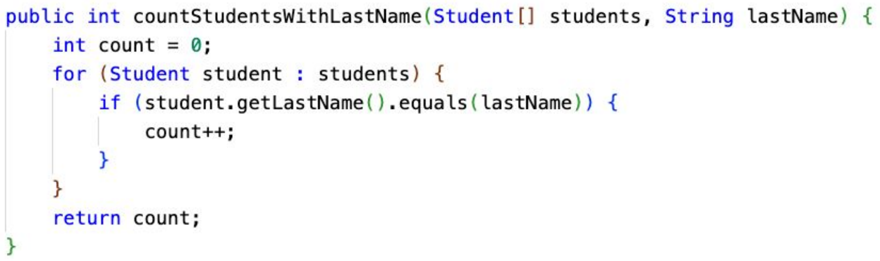
Binary search (to be reviewed on 4/28) is great for sorted arrays but won't work on an unordered one.



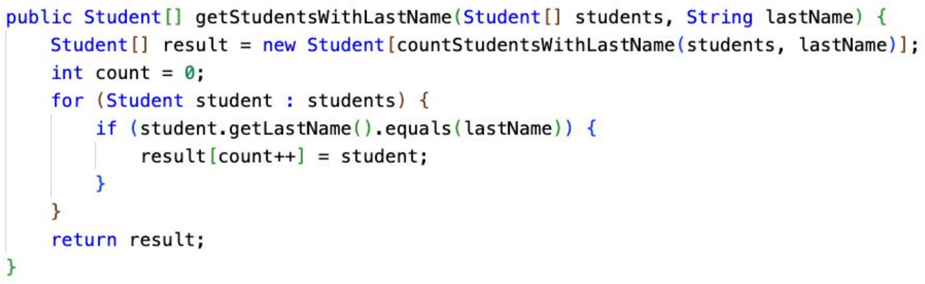
```
public Student findStudentByName(String name) {
    for (Student student : students) {
        if (student.getName().equals(name)) {
            return student;
        }
    }
    return null;
}
```

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. (Maybe we'll do this today in the FRQ...)



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



```java
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;
}
```

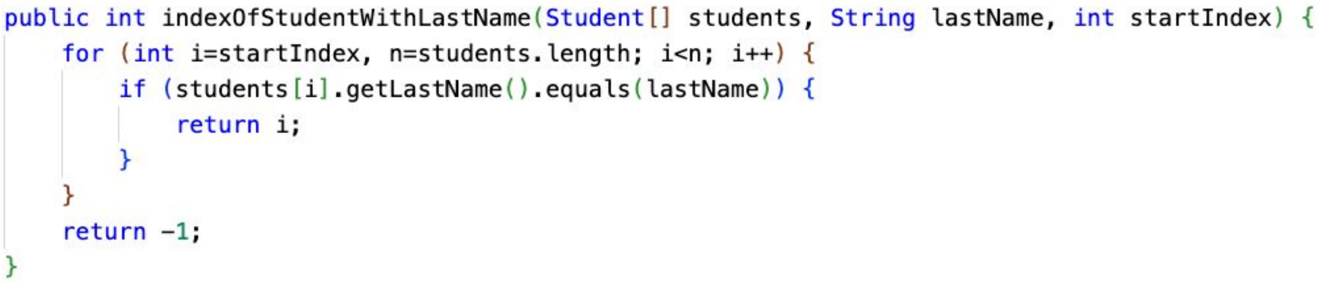
When returning an array that is a subset of the original array, you have to decide how big to make the result array.

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.

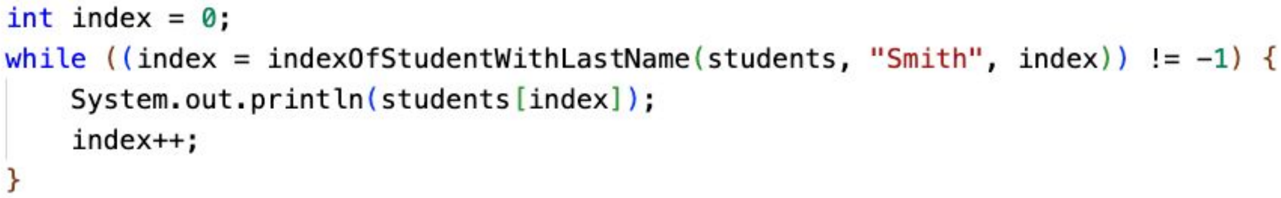
Another option would be the "growable array" pattern... In Unit 7, we'll cover ArrayList which could be used for this purpose.

Search for all matching elements

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



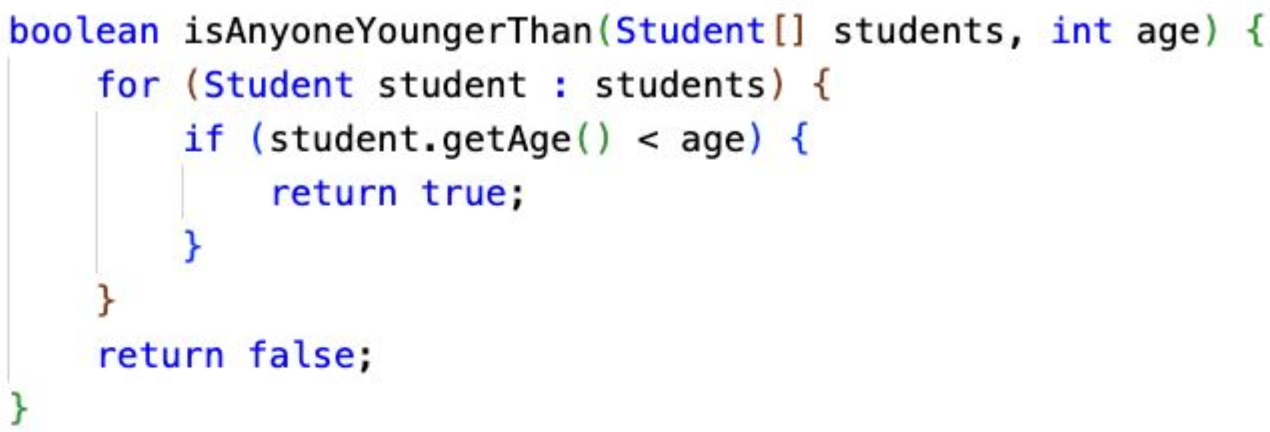
```java
public int indexOfStudentWithLastName(Student[] students, String lastName, int startIndex) {
    for (int i=startIndex, n=students.length; i<n; i++) {
        if (students[i].getLastName().equals(lastName)) {
            return i;
        }
    }
    return -1;
}
```



```java
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

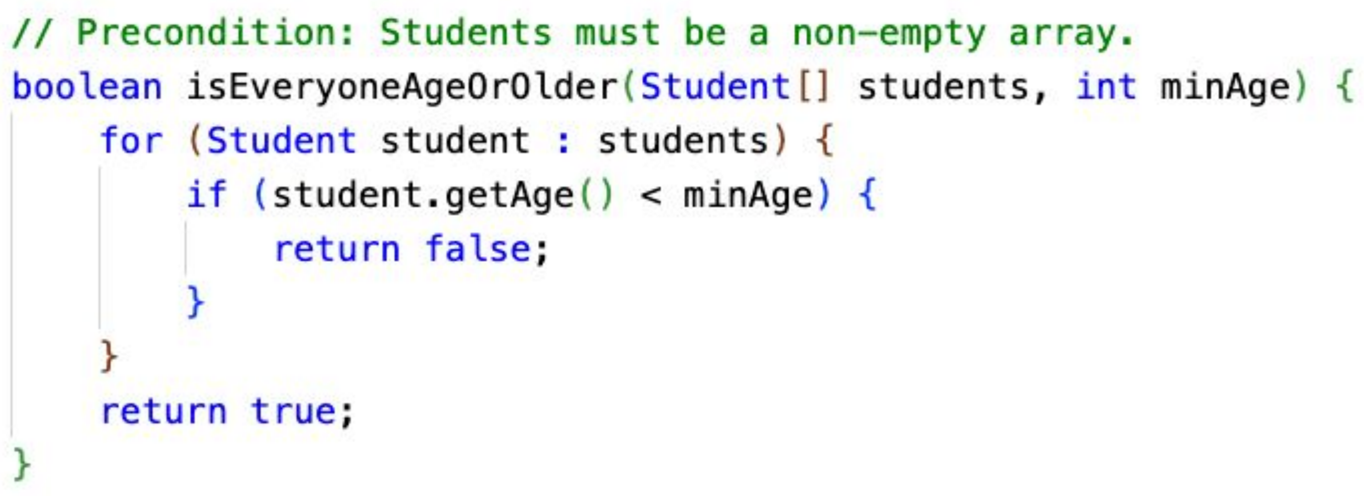
The output of this type of algorithm is a boolean. As soon as you find the first element with the desired property, you can return true.



```java
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.

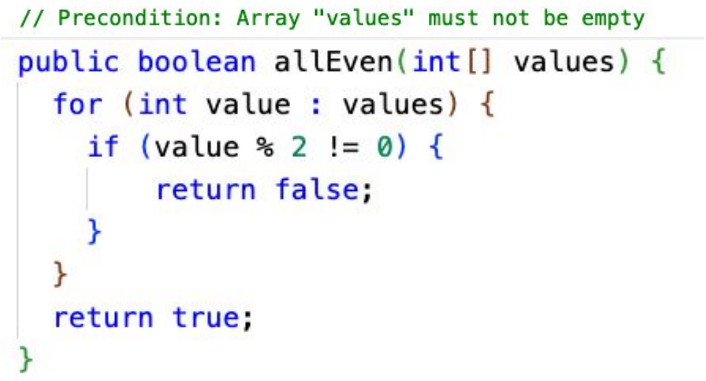


```
// Precondition: Students must be a non-empty array.
boolean isEveryoneAgeOrOlder(Student[] students, int minAge) {
    for (Student student : students) {
        if (student.getAge() < minAge) {
            return false;
        }
    }
    return true;
}
```

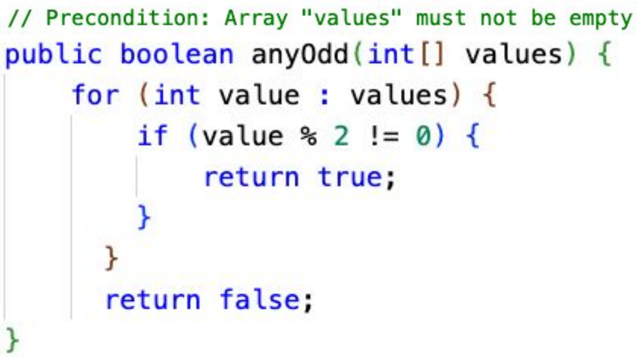
Universal / Existential Quantifiers

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

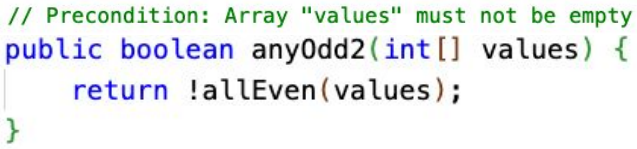
If there are any odd numbers, then not all of the numbers are even.



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

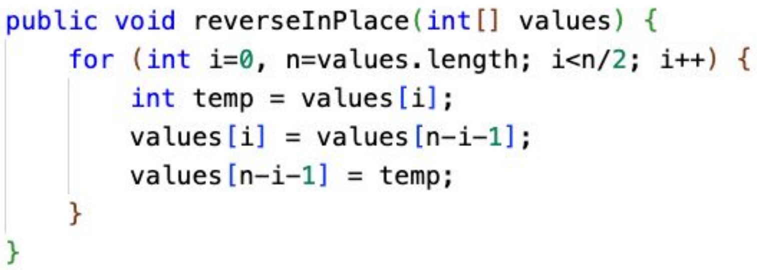


```java
// Precondition: Array "values" must not be empty
public boolean anyOdd(int[] values) {
    for (int value : values) {
        if (value % 2 != 0) {
            return true;
        }
    }
    return false;
}
```

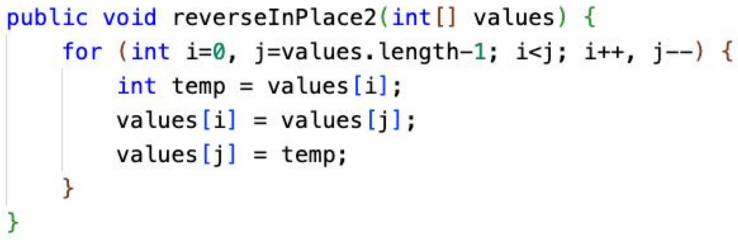


```java
// Precondition: Array "values" must not be empty
public boolean anyOdd2(int[] values) {
    return !allEven(values);
}
```

Reverse an array (in place)



```java
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;
    }
}
```



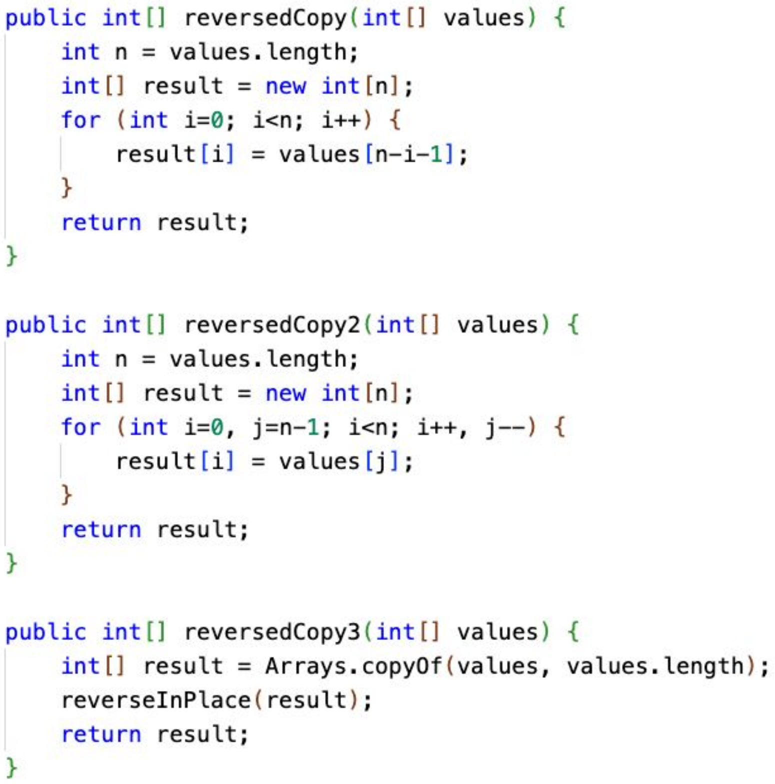
```java
public void reverseInPlace2(int[] values) {
    for (int i=0, j=values.length-1; i<j; i++, j--) {
        int temp = values[i];
        values[i] = values[j];
        values[j] = temp;
    }
}
```

The top implementation is fine, but the bottom one uses two "pointers", `i` and `j`, starting at each end of the array, 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



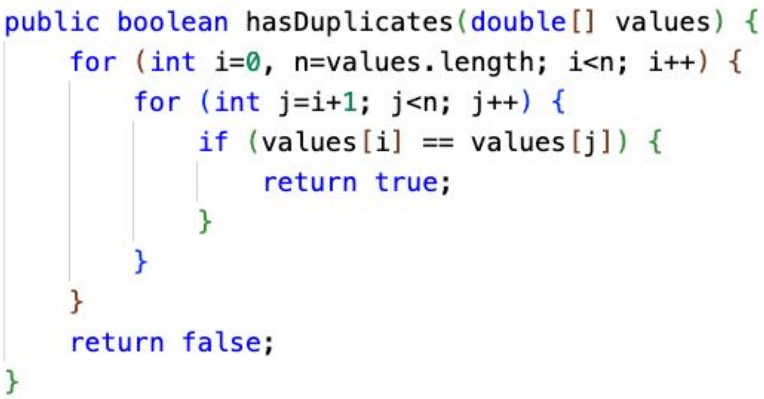
```java
public int[] reversedCopy(int[] values) {
    int n = values.length;
    int[] result = new int[n];
    for (int i=0; i<n; i++) {
        result[i] = values[n-i-1];
    }
    return result;
}

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.

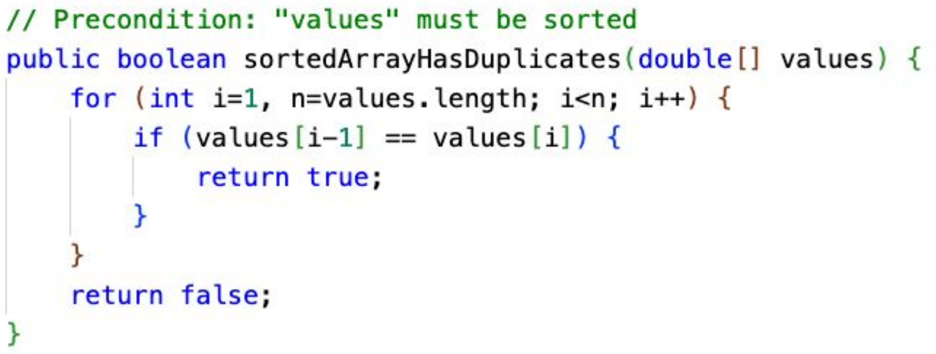
It could still be helpful to use two counters instead of one.

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



```java
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;
}
```

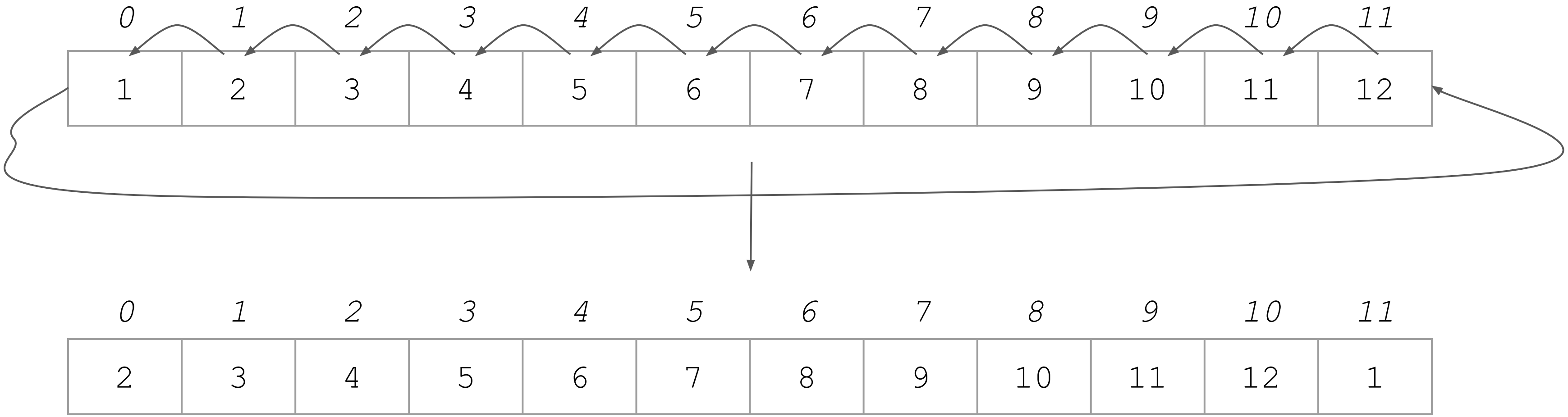


```java
// Precondition: "values" must be sorted
public boolean sortedArrayHasDuplicates(double[] values) {
    for (int i=1, n=values.length; i<n; i++) {
        if (values[i-1] == values[i]) {
            return true;
        }
    }
    return false;
}
```

Shift or rotate an array

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

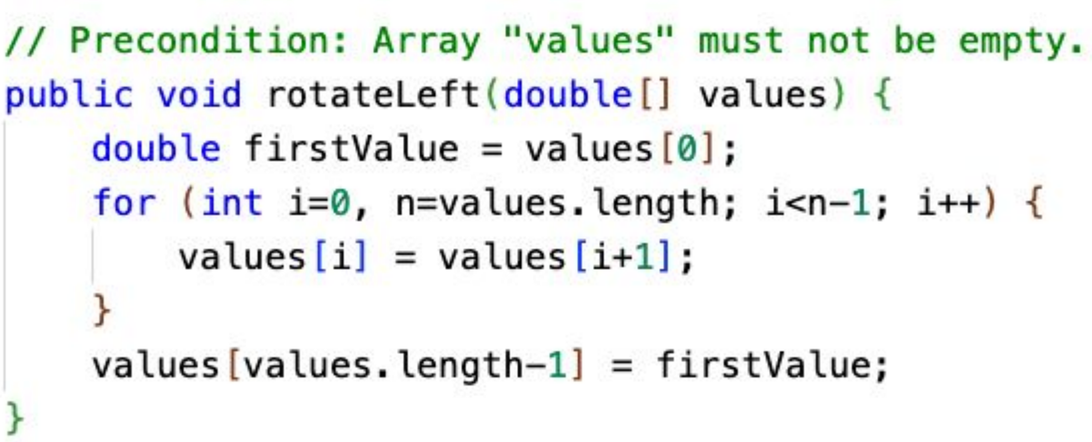
a[i] = a[i+1] for all i. The first element gets moved to the last position.



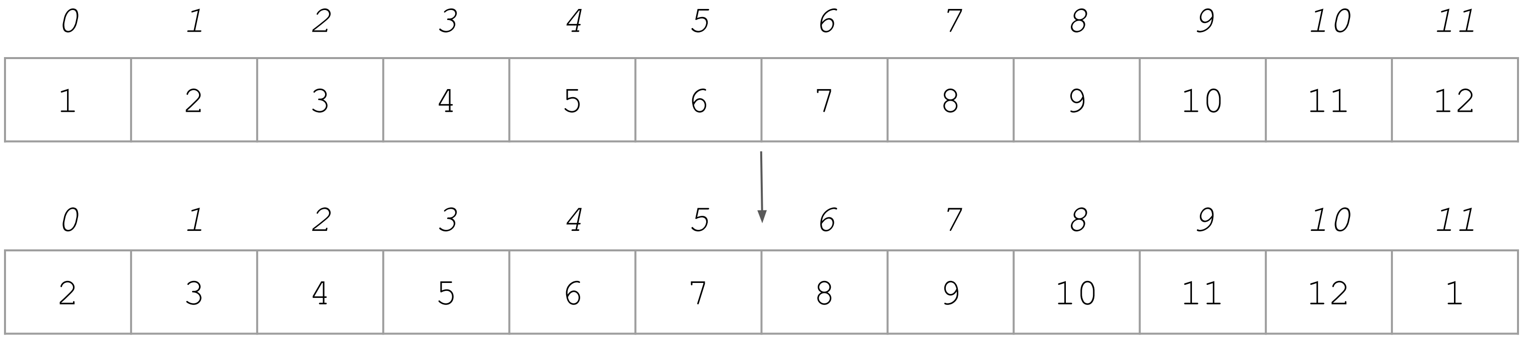
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
1	2	3	4	5	6	7	8	9	10	11	12

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

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;
}
```

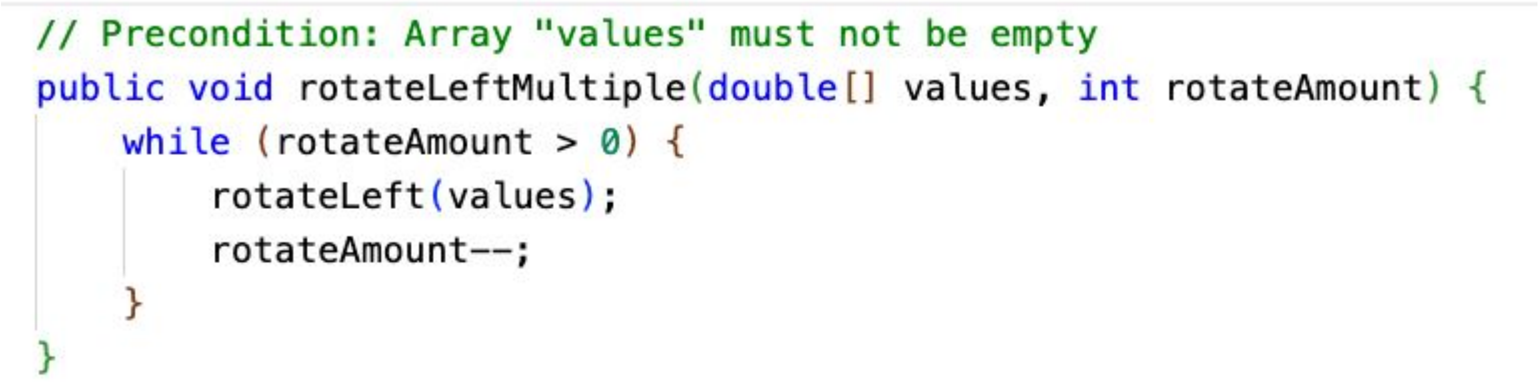


0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
1	2	3	4	5	6	7	8	9	10	11	12

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

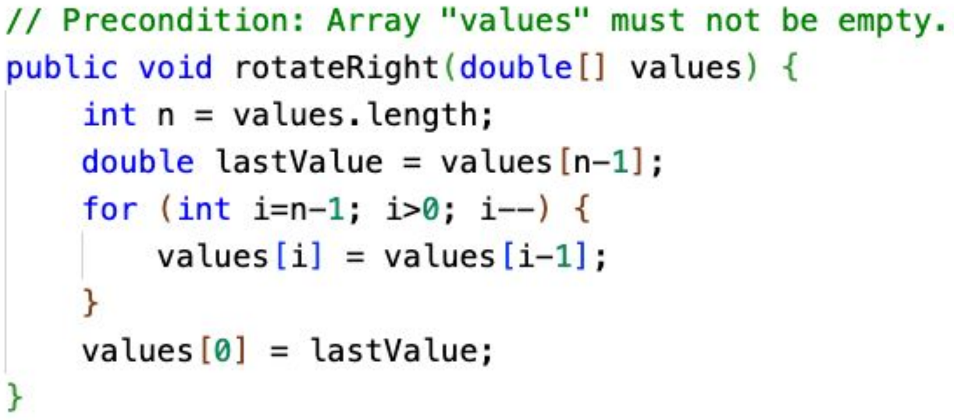
Rotate array to the left multiple positions, in place

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

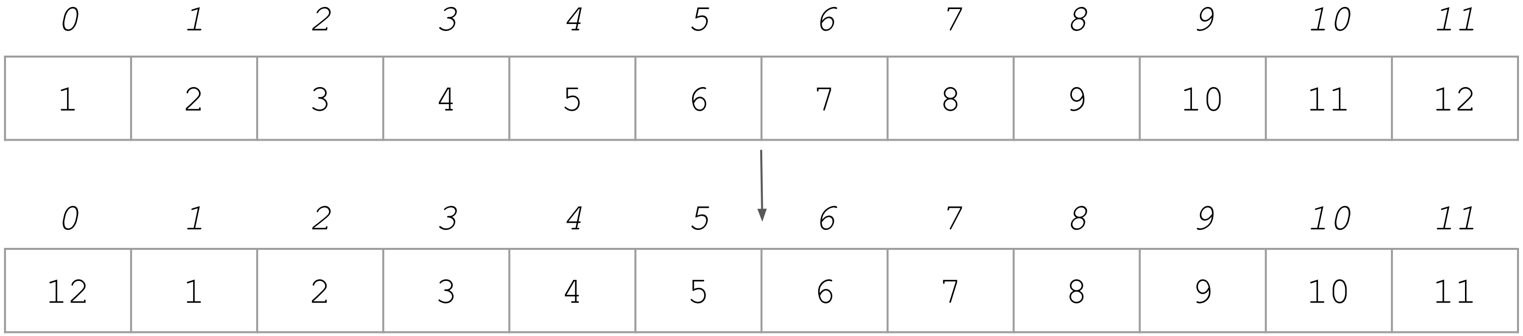


```java
// Precondition: Array "values" must not be empty
public void rotateLeftMultiple(double[] values, int rotateAmount) {
    while (rotateAmount > 0) {
        rotateLeft(values);
        rotateAmount--;
    }
}
```

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;
}
```

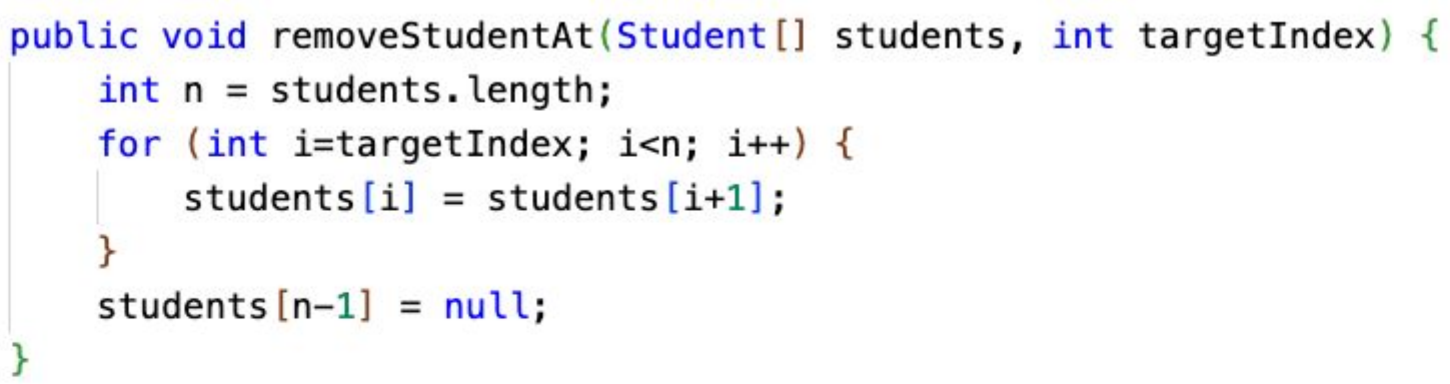


0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

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

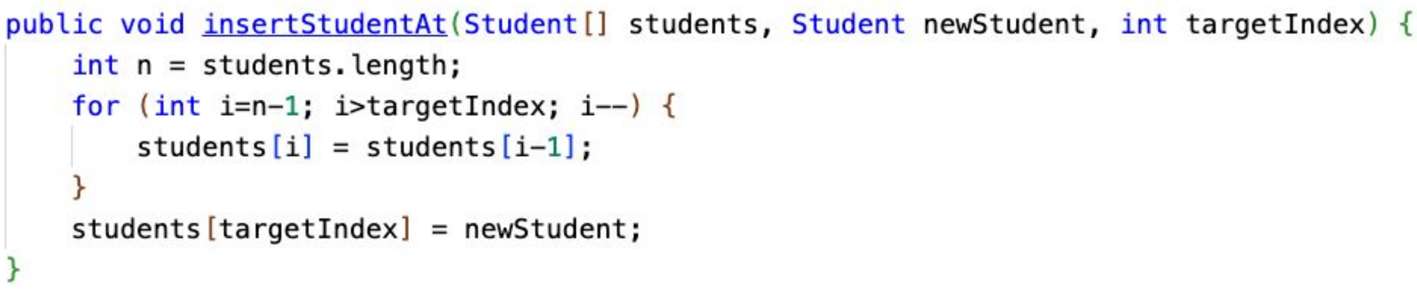
This is essentially like rotating left, but starting at a particular spot. Here, we fill in the end with **null**.



```
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."



```java
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!

AP CS FRQ 3

(25 minutes)

2022 AP Computer Science A - Free-Response Questions

Complete (3.A) and (3.B)

AP CS FRQ 3 - Review

(15 minutes)

Sample Responses and Scoring Commentary - FRQ-3