You can access to the class material from the following link:

https://drive.google.com/drive/folders/1yo4pvaTngQ1L2TkHuWPCzbfapLDyvwUt?usp=sharing

You can check your lab attendance and quiz scores from the following link:

https://docs.google.com/spreadsheets/d/1xKQEqMSGlj3RBo9kLh97UzbTO1fyCvaT/edit?usp=sharing&ouid=104851561367296854968&rtpof=true&sd=true

Lecture Notes for Week#7

Recap of Week #6:

Two important methods of Object and Overriding methods

Employee extends Object and inherits the methods of Object. There are two important methods we should override because their default behavior is not very useful.

1) String **toString()**

This method produces a String representation of the object.

```
/*change the behavior of the inherited toString */
public String toString() {
        return getNumber() + ": " + getName();
}
```

2) boolean **equals(Object o)**

The **equals** method is provided to examine the contents of the object to determine if they are structurally equal. A correct equals method that says this Employee is equal to the input if the input is an Employee with the same employee number and the same name:

```
@Override
/* Change the behavior of the inherited equals method. Two employee instances are the same if
they have the same number and name */
public boolean equals(Object o) {
    if (o instanceof Employee) {
        Employee e = (Employee)o;
        return this.getNumber() == e.getNumber() && this.getName().equals(e.getName());
     }
     return false;
}
```

instanceof: returns true if object that is the left operand can be typecast as the type that is the right operand.

Fields / Variables:

There are 4 kinds of variables, and the kind you create depends on where you create it:

1) class (static) fields: the variable is stored in the class (one copy that all the objects share). Class fields exist for the duration of the program. They are given default initial values of 0, 0.0, false, or null (not a valid address), depending on their type.
2) instance (non-static) fields: a separate variable stored in each instance of the class. Every instance has its own version. An instance field exists as long as the containing instance exists. They are given default initial values of 0, 0.0, false, or null
3) method parameter: the variable(s) that store an input to the method. It exists only in the body of the method. The initial value is set by the method call input
4) local variables: a variable declared inside a method. It exists from the moment created until the end of the compound statement it is declared it. Tthey are not given an initial value, and they must be assigned a value as their first use

Introduction to Loops

There are 4 basic loop structures in Java: while loop, for loop, do-while loop, foreach loop. The first three are general purpose loops. The fourth is a special loop form that can only be used with arrays and Iterable types. We will cover foreach loops later in the term.

1. The while loop:

```
while (condition)
  loop-body-statement
```

loop-body-statement is a single statement, simple or compound, condition is any boolean expression

While loop behavior:

1) the condition is evaluated
2) if the condition is true:
  2a) the loop body statement is executed
  2b) repeat step 1
if the condition is false, go to the next statement of the program

2. The for loop

```
for (initial statement; condition; increment statements)
        loop-body-statement
```

-the initial statement is a single statement
- increment statements are 0 or more statements, separated by commas

For loop behavior:

1) the initial statement is executed
2) the condition is evaluated
3) if the condition is true
  3a) the loop body statement is executed
    3b) the increment statements are executed
    3c) repeat step 2
if the condition is false, go to the next statement of the program

3. The do-while loop

```
do {
  loop-body-statement(s)
} while (condition);
```

<u>do-while-loop behavior:</u>

```
1) Execute the loop-body-statement(s)
2) Evaluate the condition
  If the condition is true, repeat step 1
  If the condition is false, go to the next statement of the program
```

The do-while loop should be avoided except in cases where you really need the body of the loop to execute before testing the condition.

The problem with the do-while loop is that, if you accidentally drop the "do", the rest of the code is still valid Java, but it does something entirely different! It becomes a compound statement followed by a while loop!

Some guidelines:

- for loop advantage is that the code that describes the loop is all in the header.
- for loops are good when the loop is controlled by variable(s) and there is a simple increment.
- while loop advantage is that the syntax is simpler.
- while loops are good when the increment is complicated

<u>Remember Strings:</u>

Recall that String is a class that represents text. A String object can not be changed once created. Some useful methods and operators:
+: creates a new String that is the result of concatentating two Strings
length(): returns the number of characters in the String
charAt(5): returns the 6th character of the String. (In Java, the first character is at index 0.)

IMPORTANT NOTE Creating Strings:

When building a string using a loop, we may be tempted to use the + operator.

```
String result = "";
for (....) {
  ....
  result = result + c;
  ....
}
return result;
```

However, this is not a good solution. **The + operator creates a new String each time**. If we are dealing with a very long String, such as if we want to capitalize a DNA code of millions of characters, we will be creating a LOT of unnecessary Strings and using up our memory.

Java provides a StringBuilder class to create Strings. **StringBuilder** has all the same methods as String (charAt, length, etc) plus several others. Once useful one is **append** that adds new characters (or other values) to the end of the string being created.

How to create an empty StringBuilder? Just like you create any other initial instance:
StringBuilder result = new StringBuilder();
How to add an element to the StringBuilder object? We can use append method like:
result.append(what we want to append)
What do we return at the end? We can't return result because it is not a String. But Object has a method that returns a String representation, and every class inherits it from Object.

```
        result.toString();
```

Week #7 Lecture:

SESSION-I

The Java API (Application Program Interface): https://docs.oracle.com/javase/8/docs/api/
API is typically part of any pre-defined software package that you can use in your programs. It lists how you are to use the predefined programs. The Java API lists all the pre-defined classes (and other types) in Java.
In the API you can find:
- the package the class is in (what you need to import)
- the header for the class
- a list of all non-private inner classes, constructors, fields, and methods

Suggestion: Keep the API bookmarked because we will be using it a lot in the course.

RULES FOR O-O CODING:
1. Create private fields
2. Create public getter/setter methods so that classes that extends this class can change behavior if they want.
3. Everywhere in our code that uses a value, we use the getter/setter methods instead of the fields.
4. Except in the constructor where, if we want a field to be initialized, we use the field directly.

O-O TYPE RULES:
1. Every instance is many types at the same time.
2. The "true" type is what the instance is at creation (from: new XXX() → the true type is XXX).
3. The "current" type is what the instance is typecast as at this particular part of the code.
4. An instance may only call methods and fields that exist for the "current" type.
5. However, the version of the method used is the version of the "true" type. (This applies only to instance methods! Fields and static methods are still used from the current type)
Today we will see why those rules greatly simplifies our coding:

Example: Let's create a method in Employee that compares employee's by how much money they make. We created the method **earnsMoreThan** that compares the salaries of this employee to another employee.o Suppose we have this:

```
public boolean earnsMoreThan(Employee e) {
        return this.getSalary() > e.getSalary();
}
```

There is a problem: whoever created Employee assumed that all employee's have salaries. This is not the case, but we often have situations in Java where the person creating a type does not think of every situation and makes incorrect assumptions on how it is used. Because the

earnsMoreThan method is following proper O-O coding (using the getter methods instead of the fields), it is easy for other classes to adjust so that their classes properly work.

We will have every employee type define for itself what "salary" means.
For example, an hourly employee can decide that a salary is hours worked * rate per hour.
A sales employee can decide salary is number of sales * commission, and so forth.
HourlyEmployee will "define" how salary works by overriding the salary method:

```
public double getSalary() {
  return this.getHoursWorked() * this.getHourlyRate();
}
```

Because the true type version of an instance method is always used, the hourly employee will always report its salary as the product of its hourly rate and hours worked.

- it does not matter if we typecast the hourly employee
- it does not matter if we try to set the salary of the hourly employee It just works!

BAD IDEA: Use the fields directly:

```
public boolean earnsMoreThan(Employee e) {
   return salary > e.salary;
}
```

If we did this, it would take more work to get hourly employee to work with the method. Since fields are not inherited and cannot be overridden, we are going to have to make sure hourly employee calls the inherited setSalary method everytime that its hourlyrate is changed or its number of hours worked is changed.
Not only is that going to mean changes in at least two different places, it means that if Employee changes how earnsMoreThan is computed, we will have to change HourlyEmployee!

The moral: everytime you choose to not follow the O-O guidelines, you make it more challenging for other coders to extend and use your class. So, if you want to violate the O-O guidelines, you should think carefully to make sure you have a good reason to do so.

<u>Arrays:</u>
An array is a collection of variables of the same type, stored in a contiguous chunk of memory.
We can create an array to hold variables of any type:
int[] array1 → array1 is an array that will store ints
JFrame[] array2 → array2 is an array that will store JFrames
double[] array3 → array3 is an array that will store doubles
double[][] array4 → array4 is an array that will store double[]. That is, it is an array that stores arrays that store doubles.

→We can initialize the array using the new operator and stating how many variables (elements) will be in the array:

```
array1 = new int[100];
```

array1 now stores the address for an array that contains 100 int variables.

```
array2 = new JFrame[30];
```

array2 now stores the address for an array that contains 30 JFrame variables.

→Now, to access each element, we again use the square brackets. To store a value in the first element (variable) of the array:

```
    array1[0] = 12;
```

NOTE: Remember, the we start counting from 0 in Java!

Each element of an array is a variable of the given type. So, all the rules of variables apply:
array1[2] = 3.15 ← Illegal! We are trying to store a double in a variable of type int
array1[2] = 7 ← Legal!
array1[2] = 'c' ← Legal! char is narrower than int so Java will automatically widen the 'c' to int
array1[2] = array3[2] ← Illegal! We are trying to store the value from a double variable into an int variable
array1[2] = (int)array3[2] ← Now it is legal because of the typecast.

Array disadvantage: Because the array is stored in contiguous memory, we cannot change the length (number of variables) of the array once it is created.

Array advantage: Because the array is stored in contiguous memory, we can access each element in a single step, no matter how large:

int[] a = new int[1000000];
For example, to access a[95436], we do not need to run through the array to the 95436'th entry. We know the address of a (it is stored in the variable a), we know the byte size of the type of the array (for example, each int is 4 bytes) and we know which one we want (index 95436). So the address of a[95436] is: (address of a) + 4 * 95436

<u>**Loops and Arrays:**</u>
How do we fill an array? → Usually with a loop.
JFrame[] frames;   // frames will store (the location to) an array that stores JFrames
frames = new JFrame[30];   // now frames has the address of a chunk of memory divided into 30 variables of type JFrame.
But no JFrame is yet stored in frames. To place a separate JFrame in each one:

```
for (int index = 0; index < frames.length; index = index + 1) {
        frames[index] = new JFrame();
}
```

If you have a small array, then you don't need a loop. You can enter the elements on at a time, or you can use an "array constructor" shortcut.

Our first method with arrays: Create a method that reverses the contents of an array**.** Each slot in the array is just a variable and so we will use variable assignment. We will need one extra variable to store values so we do not lose any values.

1. save the value in the back of the array, array[array.length - 1 - index]

2. store the value at the front of the array array[index] into the back of the array array[array.length - 1 - index]
- this overwrites the value in array[array.length - 1 - index]. Good thing we saved it!
3. store the saved value into array[index]

When do we stop the loop? You might be tempted to say when index reaches the end (array.length), but that is not correct. Can you see why?

```
public static void reverse(int[] array) {
        for (int index = 0; index < length / 2; index = index + 1) {
                int save = array[array.length - 1 - index];
                array[array.length - 1 - index] = array[index];
                array[index] = save;
          }
}
```

More on Arrays: "increasing" the size of an array
Create a method that takes an int array and an int. The array is full, but we need to put the int value at the end of the array. We can't change the size of an array, so instead, we must create a brand-new array, copy the values over, and then put the new value at the end.

```
public static int[] append(int[] array, int x) {
        int[] newarray = new int[array.length + 1];
        for (int i = 0; i < array.length; i = i + 1)
                newarray[i] = array[i];
        newarray[newarray.length - 1] = x;
        return newarray;
}
```

First, note that we needed a loop to do the copy. If we write "newarray = array;" we instead copy the address of array to newarray. That has the effect of having both newarray and array point to the same array.
Second, note that we copied x in outside the loop. We could have tried to also include x inside the loop, but that would mean an if statement and more complicated code. To keep your code simple, remember to have each loop accomplish one task.
Third, note that we had to return newarray. If we did not, we would lose the newarray. newarray is a local variable and so it is lost once the method ends. When the method ends, all local variables and method parameters are removed from memory. If we lose newarray, we lose the address to the new array. By returning the new address, we can have it outside the method.

Another incorrect assumption is to have an assignment statement: array = newarray. This would copy the address of newarray to array, but it has the same problem because array is a method parameter. Thus, array will be lost once the method ends.
Example:
myArray = append(myArray, 11); → will create a new array one larger than myArray, copy the data over, add 11 to the end, and then return the address of the new array, and that address is stored in myArray. myArray now points to a new array one larger than the array it originally

pointed to. The old array is still in memory. Java will eventually de-allocate it if there are no other variables storing its address.

SESSION-II

Arrays and Typecasting: We can always typecast an array to Object because everything that is not a primitive value in Java is an object.
int[] array1 ← array1 will store the location of the array in memory
new int[100] ← the new operator will return the address of the array
(Object)array1 ← the typecast is legal because Object is above everything in the class hierarchy.

Typecasting the individual variables in the array: Because a typecast on a non-primitive type does not change the true type of the object, just its current type, Java allows us to typecast the types of the variables for arrays of non-primitive types:
JFrame[] frameArray = new JFrame[100]

1- Object[] oArray = (Object[])frameArray; ← Legal. Object is wider than JFrame. An explicit typecast is not needed.
2- frameArray = (JFrame[])oArray; ← Legal as long as the true type of oArray is an array storing JFrames or something below JFrame in the hierarchy
oArray[0] = new JFrame();
3- oArray[1] = "Hello" ← Illegal. While Object o = "Hello" is legal, here oArray[1] is referring to an element of frameArray. frameArray knows that it is only to store JFrames.

This does not work for primitive arrays:
int[] array = new int[10];
1- double[] darray = (double[])array; ← Illegal. Even though double is wider than int, Java will not allow this.
char[] letters = new char[100];
2- short[] values = (short[])letters; ← Illegal. Even though char and short are the same byte size, Java will not allow this.

Java forbids any typecasting between primitive array.

Linear Searching in an Array:
Here is a standard loop for searching for an item in an array:

```
public static int linearSearch(int x, int[] array) {
        for (int i = 0; i < array.length; i++) {
                if (x == array[i])
                        return i;
        } // at this point i = array.length so we checked all entries of array
        return -1;
}
```

What is the subgoal of the loop: It is what must be true after each iteration for the loop to continue: "x is not in the first i elements of array"

Is this the best search method we can write?
In the worst case, how many array locations does it inspect? →All of them. So, in the worst case, the time the method takes will be proportional to the length of the array.
On average, it will need to check half the elements in the array. So, in the average case the time the method takes is also proportional to the length of the array.

Can we write a faster loop? No! What is the proof? No matter how we decide to run through the array, until we look at every location, we can not be sure if x is not in the array. So, unless we know more information about the array, our linearSearch method is optimal.

Searching in a sorted array: Suppse the array is sorted in non-decreasing order. Now can we write a faster method? Proposed Algorithm (Binary Search)

1. Examine the middle element.
2. If the middle element is smaller than x, repeat on the upper half of the array.
3. If the middle element is larger than x, repeat on the bottom half of the array.

This algorithm looks more complicated to code, but is it fundamentally faster? We start with a list of n elements, and each time, we cut the number of elements we are considering in half.
n → n/2 → n/4 → n/8 → ... → 1
So, the number of iterations (and the number of elements of the array) is k where n / (2^k)) = 1.
k = log(base 2) n

How much an improvement is this over linear search?

Array size	#elements linear search check	#element binary search check
8	8	3
1000	1000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30
1,000,000,000,000,000	1,000,000,000,000,000	50

So, it is very worth it to write this method. Here is the first attempt we made in class. We are using a while loop because we do not know, at first, how many times we will need to iterate.

Binary Search, More on Reasoning About Loops
Suppose the array is sorted in non-decreasing order. Now can we write a faster method?
Proposed Algorithm (Binary Search):

1. Examine the middle element.
2. If the middle element is smaller than x, repeat on the upper half of the array.
3. If the middle element is larger than x, repeat on the bottom half of the array.

Here is our first incorrect attempt:

```
/** Return the index of x in a or -1 if x is not in array.
* Precondition: a is sorted in non-decreasing order */
public static int binarySearch(int x, int[] array) {
    int front = 0;       // stores the smallest index of array in the region still being considered
    int back  = array.length - 1; // the largest index of array in the region still being considered
    while (front != back) {
        int mid = (front + back) / 2;
        if (x < array[mid])
            back = mid;
        else if (x > array[mid])
             front = mid;
        else // only other case is x == array[mid]
             return mid;
     }
     return -1;
}
```

Loops are tricky to get completely correct so it is not surprising that our first attempt is not quite correct. Because of that, computer scientists have developed mathematical logic to reason about loops.

First, we have to decide what the goal of our loop is. That is something we -should- be able to do since we are writing the code. (Note, this is another reason why we want our loops to do just one task. It makes describing the goal easier.)

Given the goal for the loop, we must next determine what the -subgoal- for each loop iteration is. The subgoal (in formal math terms the subgoal is called the "loop invariant") is what we need to accomplish with each iteration of the loop in order to achieve the goal. Verifying that a loop is correct means that we verify that we are able to achieve the subgoal after each iteration, and we verify that the loop will eventually stop.

A correct subgoal has the following properties:

a) The subgoal is true after each iteration of the loop.

b) The subgoal is true just before the first iteration of the loop. (Technically, this can be thought of as the same as (a) by saying that if you have not entered the loop, then you are at the end of the "0th" iteration.)

c) The subgoal AND the loop condition being false logically implies the goal of the looo.

Let us check the reasoning of our loop.

1) What is the goal: At the end of the loop, x should not be in array. (Because if we reached the end of the loop, we did not return x.)

2) What is the subgoal for each loop iteration: After each iteration, if x is in array, it is in array[front...back] (i.e. it is between front and back, inclusive).

Now let us verify the subgoal. Recall the three properties we need:

a) The subgoal is true at the end of each loop iteration.

b) The subgoal is true immediately before the first iteration of the loop starts.

c) When the loop condition becomes false, the subgoal and the condition being false logically implies the goal.

Property b is easy to verify. Before we start the loop, front = 0 and back = array.length - 1. Of course, if x is in array it will be in array[0..length-1].

Property c is does not hold in our loop: When the loop condition becomes false, the subgoal and the condition being false implies the goal. The loop condition becomes false when front == back.

The subgoal is now: "If x is in array, it is in array[front...front]"
That does NOT logically imply the loop goal! x could still be in array if x is at array[front].

Using the logic reasoning, we see that our loop is incorrect. Maybe thorough testing would have found this problem, but the error will only occur in specific situations (do you see why?). Let us fix the method by changing the loop condition so that when it is false we do get the loop goal:

```
while (front <= back) {
```
Now, when the loop stops, back < front and so the loop subgoal becomes "if x is in array, it is in array[front..(front-1)]", and an array where the first index is larger than the last index is mathematically empty.

Next, we will check point (b): the subgoal must be true after each iteration. The way you verify this is check that if the loop subgoal is true at the start of the iteration, then it is still true at the end of the iteration. See if you can reason through the code and verify that we always achieve the loop subgoal.

Are we done? We verified the loop subgoal so the logic of our loop is correct, but we still must verify that the loop terminates. Having a logically correct loop does no good if the loop runs forever.
Are we guaranteed that the loop will terminate, or could it run forever? To guarantee that it does not run forever, we have to show that the gap between front and back decreases with each iteration. Thus, either front increases or back decreases. Is it possible to have a situation where front and back to do not change? YES!! Consider front = 5 and back = 6 and the element we are looking for is at index 6. Note that in this situation, mid = 5, array[mid] is smaller than the element, and so front = mid = 5. front never changed! **What is our fix?** To note that we know that the element is not at the array[mid] and so we will not include it. Since we know that front <= mid <= back, setting front = mid + 1 and back = mid - 1 guarantees that either front is increased or back is decreased on each iteration.

Here is the corrected code:
```
/** Return the index of x in a or -1 if x is not in array.
* Precondition: a is sorted in non-decreasing order */
public static int binarySearch(int x, int[] array) {
        int front = 0;   // stores the smallest index of array in the region still being considered
        int back = array.length - 1;  //the largest index of array in the region still being considered
        while (front <= back) {
                int mid = (front + back) / 2;
                if (x < array[mid])
                        back = mid - 1;
                else if (x > array[mid])
                        front = mid + 1;
                else // only other case is x == array[mid]
```

```
                        return mid;
        }
        return -1;
}
```

The moral: using the formal logic reasoning helps us design correct loops before we code them into the program.

Loop Testing

How to test loops?
Here is a good "rule of thumb" to follow to help you catch most of the usually bugs that can show up in loops:

1) Test 0, test 1, test many
2) Test first, test middle, test last

Consider the binarySearch method.
What does "test 0, tests 1, test many" mean?
One interpretation is test arrays of length 0, 1, and many. What does "test first, test middle, test last" mean? One interpretation is to test where we are searching for an element in at the front of the array, an element at the back, and an element somewhere in the middle.

Another interpretation is to test where we find the element quickly (the first iteration of the loop), and one where the loop runs to the end (the element is not in the array). So, this guide suggests that we have to test the following array lengths: 0, 1, and large, and we have to search for the middle element, elements not the middle, the element at the front, the element at the end, and elements that are not in the array: one smaller than all elements in the array, one larger than all elements in the array, and one that falls between the largest and smallest.

Week#8 Lecture Sneak Peek

Designing Good Loops, part 1.

Loop example: Write a method that determines if an English string is a palindrome. We will say that a string is a palindrome if

1) non-letters are ignored
2) letter capitalization does not matter
3) the remaining characters read the same forwards as backwards
ex: "Madam, I'm Adam."

This problem is a little harder than the ones done previously. We will first look at some possible algorithms, but it turns out that the best, and simplest, algorithm is not going to be the first ones considered. Often, it is a good idea to think through the benefits and disadvantages of different possible algorithm and to try to improve the algorithm before writing code.

Here are three algorithms proposed (or slight variations of the ones that were proposed) by students from the previous years.

Proposed algorithm 1:

```
1) Create a new String (or StringBuilder) that removes all punctuation
2) Create a new String (or StringBuilder) that removes all spaces
3) Make every letter upper case
4) Do the isPalindrome logic or method from last class
```

Proposed algorithm 2:

```
1) Create a new String (or StringBuilder) that contains only the letters from the input string
2) Do the isPalindrome logic or method from last class
```

Proposed algorithm 3:

```
1) have two indeces, one at the front and one at the back
2) Repeat:
        2a) Repeat: until the first index points at a letter, increment it
        2b) Repeat: until the second index points at a letter, decrement it
        2c) Compare the letters at the two indeces, and return false if they don't match and
increment/decrement the indeces otherwise
```

What we want for a good algorithm:
- **Time and space efficiency**: Do not use more memory that is needed, do not make more traversals of the String than needed. We have to be a little careful when counting traversals because two traversals that do one operation at each step is the same as one traversal that does two operations in each step.
- **Logical simplicity:** The loop should do one logical task. If we need two different tasks, we should use separate loops for each.
- **Simple iterations:** The loop should do a single iteration with each step.

Problems and advantages of algorithm 1 and 2:
- The algorithms are simple with each loop doing one task.
- The algorithms create extra, unnecessary Strings.
- The algorithms use multiple traversals of the String.

Algorithms 1 and 2 are very similar. Here is code for algorithm 2:

```
StringBuilder b = new StringBuilder();
        for (int index = 0; index < s.length(); index = index + 1) {
                if (Character.isLetter(s.charAt(index))
                        b.append(Charater.toUpperCase(s.charAt(index)));
        }
        return isPalindrome(b.toString());
```

How much extra memory did it use? It created one new StringBuilder plus one new String (the toString method). So, it needed more memory equal to twice the length of the string. (Note that if we used the + operator instead of StringBuilder, we would need more memory that is equal to the square of the length of the string!)

Hint for writing good algorithms:

1- Think about how you would solve the problem.
2- We usually hate doing unnecessary busy work and so our natural solutions avoid a lot of the "extra memory" isses or "extra traversal" issues of the previous solutions.
3- If we had to solve this problem using pencil and paper, we would never waste time first copying down the string again without the punctuation, and then copying another time changing the letters to lower case.

One important skill is to learn how to take what we naturally do in problem solving and then breaking that down into simple steps that you can write into a program.

Problems and advantages of algorithm 3:
The algorithm does not create any extra space and does one traversal of the string
Each inner loop does 1 task, and the outerloop then does essentially one task.
However, the increment, from the point of view of the outer loop, is complex because the indeces will change by arbitrary amounts each iteration

```
for (int front = 0, back = s.length() - 1; front < back; front = front + 1, back = back - 1) {
        while (!Character.isLetter(s.charAt(front))       // repeat until front hits a letter
                front = front + 1;
        while (!Character.isLetter(s.charAt(back)) // repeat until back hits a letter
                back = back - 1;
          // now test the two characters
        if (Character.toUpperCase(s.charAt(front)) != Character.toUpperCase(s.charAt(back))
                return false;
        }
}
```

BUT THIS CODE IS NOT RIGHT!!! IT WILL CRASH
We need to add extra conditions in the each while and in the test!
We have to make sure each while does not move the index outside the string or s.charAt(..) will return an error. We need to make sure, when we test the two characters after the while loop, that we are testing two letters. Maybe either while loop halted because we reached the end of the string.
The point is: because we do not have all the loop increment information inside the for-loop header, the writing the algorithm has become much more complicated. Recall, though, that this code did have the advantage of a single traversal and no extra memory created. So, a good algorithm will keep those features but do only a simple increment at each step

Here is a better algorithm that keeps the same general logic as algorithm 3 but uses simple increments:
1) Have two indeces, one at the front and one at the back.
2) Repeat until done:
  2a) If the first is not a letter, move the index up one and repeat 2
  2b) Otherwise if the second is not a letter, move the index down one and repeat 2
  2c) Otherwise if both matches, move both indeces one character
  2d) Otherwise we have letters that do not match and we can return false.
- This algorithm uses no extra space and does a single traversal of the String.

- The algorithm is logically simple because we are still doing one task: comparing each letter at the front to the corresponding letter at the rear.
- The loop has a simple organization because each index will increment or decrement AT MOST once each iteration, and at least one will change.
- Since either the front or end index changes, it is easy to logically reason that our loop will eventually terminate.

```
public static boolean isEnglishPalindrome(String s) {
        int front = 0;
        int back = s.length() - 1;

        while (front < back) {
                if (/* the front is not a letter */)
                        front = front + 1;
                else if (/* the back is not a letter */)
                        back = back - 1;
                else if (/* the two letters do not match */)
                        return false;
                else {
                        front = front + 1;
                        back = back - 1;
                }
        }
        return true;
 }
```

To check if things are letters and to capitalize, etc, we can either write the math code from last lecture or we can use the API. There is no difference in terms of simplicity or efficiency.

```
public static boolean isPalindrome(String s) {
   int front = 0;
   int back = s.length() - 1;

   while (front < back) {
      if (!Character.isLetter(s.charAt(front)))
         front = front + 1;
      else if (!Character.isLetter(s.charAt(back)))
         back = back - 1;
      else if (Character.toUpperCase(s.charAt(front)) != Character.toUpperCase(s.charAt(back))
         return false;
      else {
         front = front + 1;
         back = back - 1;
      }
   }
   return true;
}
```