

10/24/22

*A few notes on*  
Shakespeare

# State Machines - Regular Expressions

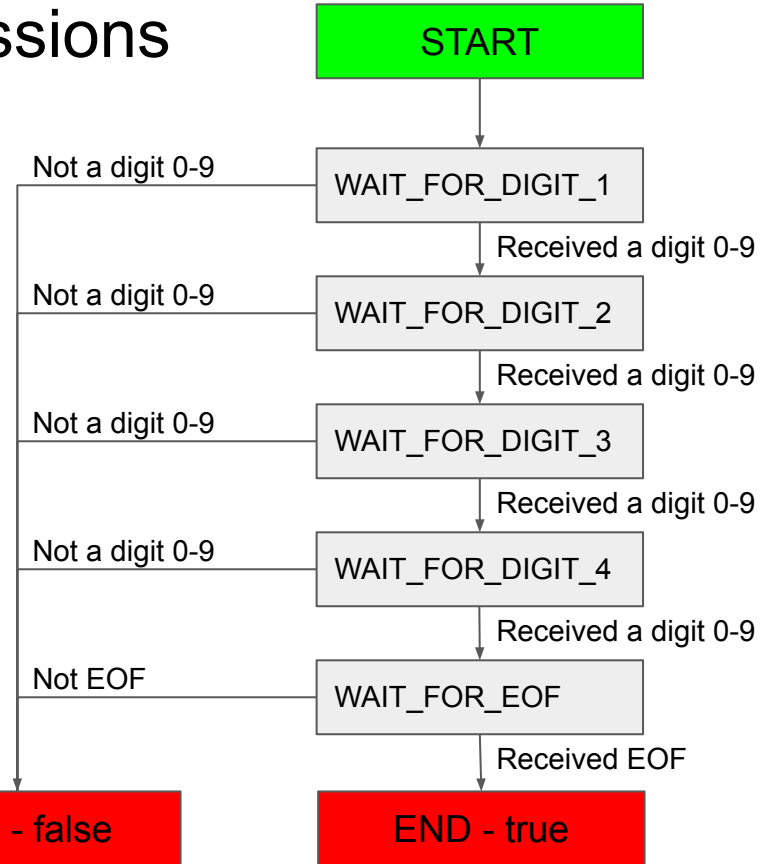
**\A\d{4}\z**

```
import java.util.regex.*;
class Main {
    public static void main(String[] args) {
        Pattern yearPattern = Pattern.compile("\\A\\d{4}\\z");

        Matcher m = yearPattern.matcher("1609");
        System.out.println(m.matches());

        m = yearPattern.matcher("Foobar");
        System.out.println(m.matches());

        m = yearPattern.matcher("16309");
        System.out.println(m.matches());
    }
}
```



# Unit 4 Review

Chapter 4.6

# 4.1

The while loop

# Iteration

**Iteration**, in the context of computer programming, is a process wherein a set of instructions are repeated a specified number of times or until a condition is met.

Each time the set of instructions is executed is called an **iteration**.

Another term for iteration is **loop**... the program "loops back" to an earlier step and repeats.

# while syntax

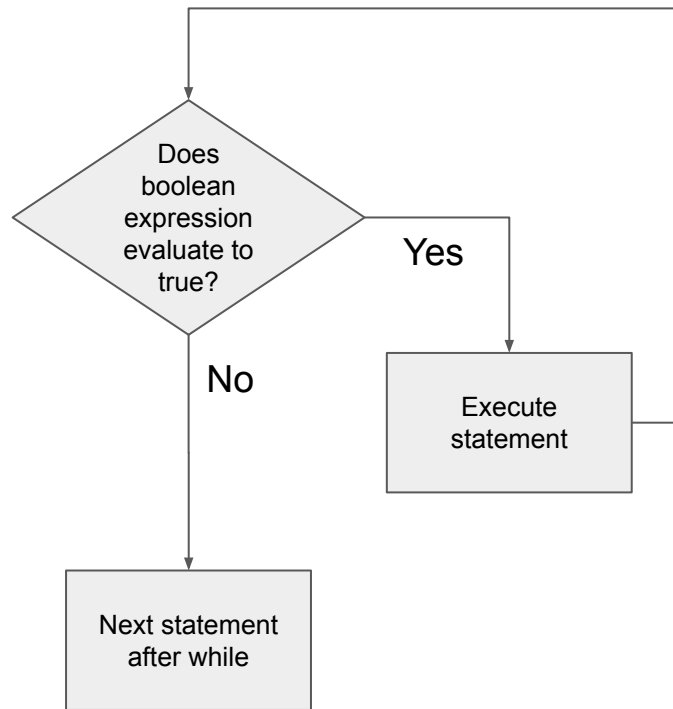
```
while (boolean expression)  
    statement
```

Example:

```
int i = 1;  
while (i <= 100) {  
    System.out.println(i);  
    i++;  
}
```

It's like an if statement that keeps repeating itself.  
Like if, we recommend curly braces always.  
(But Java does not require it.)

Flowchart of while



# ++ and -- operators

```
int i = 1;
while (i <= 100) {
    System.out.println(i);
    i++;
}
```

```
void printManyTimes(String s, int count) {
    while (count > 0) {
        System.out.println(s);
        count--;
    }
}
```

```
int i = 1;
while (i <= 100) {
    System.out.println(i++);
}
```

```
void printManyTimes(String s, int count) {
    while (--count >= 0) {
        System.out.println(text);
    }
}
```

`i++` means post-increment, so it evaluates to the current value of `i`, then increments it. The increment/decrement is a **side effect**.

Opinions differ on style here.



## = (assignment) can be used in the condition

Your loop condition may depend on some code that repeats every iteration:

```
String command = getNextCommand();
while (command != null) {
    executeCommand(command);
    command = getNextCommand();
}
```

The assignment operator can be used in the condition to avoid the repetition:

```
String command;
while ((command = getNextCommand()) != null) {
    executeCommand(command);
}
```

Programmers may differ on the style here. (The first style was used in Magpie.)

# Sentinel values

A sentinel value is a special value that tells the loop to terminate.

The code from the previous slide is an example of this:

`getNextCommand` returns null when there are no more commands to execute.

```
String command;
while ((command = getNextCommand()) != null) {
    executeCommand(command);
}
```

# Input-controlled loops

```
Main.java x +
1 import java.util.Scanner;
2
3 /**
4  * A simple class to run the Magpie class.
5  * @author Laurie White
6  * @version April 2012
7  */
8 public class MagpieRunner3 {
9     /**
10      * Create a Magpie, give it user input, and print its replies.
11      */
12     public static void main(String[] args) {
13         Magpie3 maggie = new Magpie3();
14
15         System.out.println (maggie.getGreeting());
16         Scanner in = new Scanner (System.in);
17         String statement = in.nextLine();
18
19         while (!statement.equals("Bye")) {
20             System.out.println (maggie.getResponse(statement));
21             statement = in.nextLine();
22         }
23     }
24 }
```

The while loop is often used for **input-controlled loops**, where **input** is being taken from the user, or from a file on disk, or the network.

An example is the MagpieRunner, where the while loop continues until the user enters "Bye", the sentinel value which tells Magpie that no more input is coming and terminates the loop.

# Tracing loops

```
1 ▼ class Main {  
2 ▼   private static int factorial(int x) {  
3     int result = 1;  
4 ▼   while (x > 1) {  
5     result *= x--;  
6   }  
7   return result;  
8 }  
9 ▼ public static void main(String[] args) {  
10   System.out.println(factorial(5));  
11 }  
12 }
```

x	result
5	1
4	5
3	20
2	60
1	120

# do...while syntax

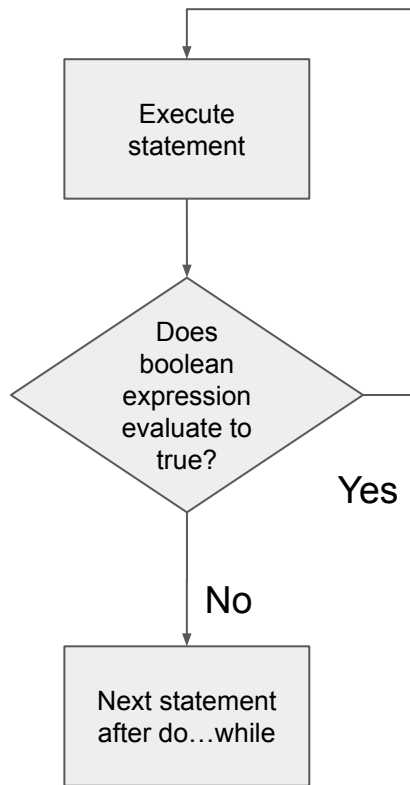
```
do  
    statement  
while (boolean expression)
```

Sometimes, you want to check some condition AFTER the body of the loop has run, not before.

Example:

```
String name;  
do {  
    System.out.println("Enter your name.");  
    name = scanner.nextLine();  
} while (name.length() == 0);
```

Flowchart of while



# Infinite loops

```
void serveRequestsForever() {  
    while (true) {  
        handleNextRequest();  
    }  
}
```

This may seem strange, but it has its place.

Sometimes the loop isn't really infinite, but the termination condition of the loop is complicated. There are ways to break out of a loop, even an infinite one (break, return).

There may also be **unintentional** infinite loops in your code that you need to fix!

# 4.2

The for loop

# Counter-controlled loops

We looked at input controlled loops, which are often done using while.

For statements are often used to do **counter-controlled loops**, where the loop is repeated a specific number of times, and a numeric counter is used to track which iteration the loop is on.

However, really, any of the loop statements in Java can be used to write any possible program. Which loop to use is a matter of what you think best expresses the intent of the program.



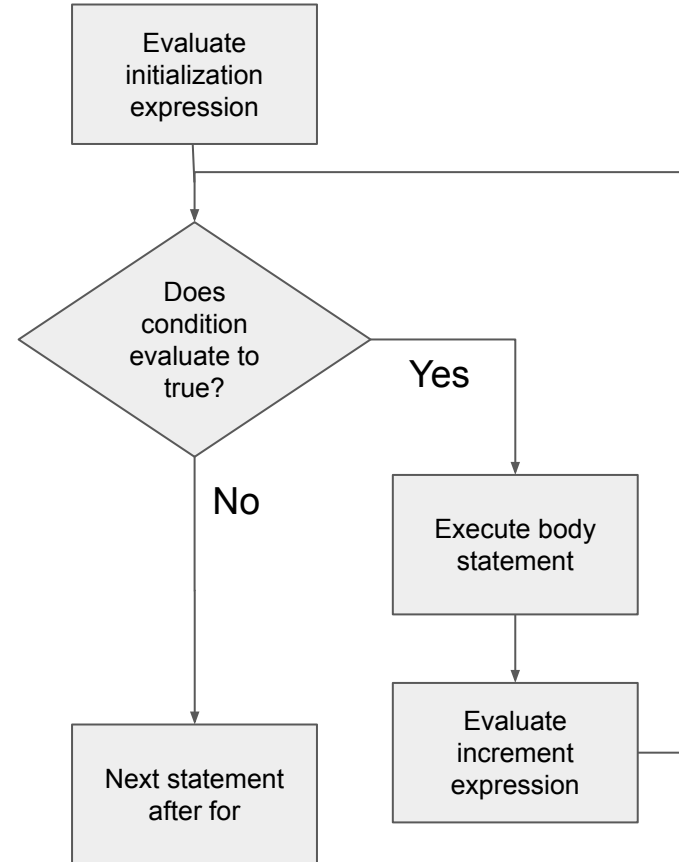
# for syntax

`for` (*initialization; condition; increment*)  
*statement*

Example:

```
int i;  
for (i = 1; i <= 100; i++) {  
    System.out.println(i);  
}
```

Flowchart of for



# for syntax

`for` (*initialization; condition; increment*)  
*statement*


*initialization* may also declare variables, even multiple variables (but only of the same type)  
The scope of any variable declarations is purely the for loop itself.

```
1 ▼ class Main {  
2 ▼   public static void main(String[] args) {  
3       String s = "Hello, world!";  
4 ▼   for (int i=0; i<s.length(); i++) {  
5       System.out.println(s.charAt(i));  
6   }  
7 ▼   for (int i=0, n=s.length(); i<n; i++) {  
8       System.out.println(s.charAt(i));  
9   }  
10  }  
11 }
```

# for syntax

The *increment* expression can use "," to update multiple variables.

And you're not limited to ++, --, although they are the most common things to see there.

A screenshot of a Java IDE window titled 'Main.java'. The code defines a class 'Main' with a 'main' method. Inside the 'main' method, a 'long' variable 'value' is initialized to 1. A 'for' loop is used with 'int i = 0' as the initialization, 'i < 64' as the condition, and 'i++, value \*= 2' as the update expression. The loop body contains a 'System.out.println' statement that prints the value of 2 raised to the power of 'i'.

```
1 class Main {  
2     public static void main(String[] args) {  
3         long value = 1;  
4         for (int i = 0; i < 64; i++, value *= 2) {  
5             System.out.println("2^" + i + " = " + value);  
6         }  
7     }  
8 }
```

# for syntax

```
for (initialization; condition; increment)  
    statement
```

***initialization***, ***condition***, and ***increment*** are all optional

```
for ( ; ; ) { ... }
```

is an infinite loop, the same as `while (true) { ... }`

## Why would you omit initialization?

Sometimes the initialization needed takes multiple statements and can't be easily expressed in a single expression, so you do it before the for loop and omit the initialization.

## Why would you omit increment?

Increment logic may similarly get complicated and be better expressed within the body of the loop.

# for syntax

```
for (initialization; condition; increment)  
    statement
```

for statements are very frequently used with numeric counters, usually integers. But they don't have to be... the expressions can be most anything.

```
// Cast spell to magically look east as far as possible.  
for (Room room = player.getLocation();  
     room != null;  
     room = room.getEast()) {  
    printRoomContents(room);  
}
```

**for** is very flexible in this way.

Some languages like BASIC have a FOR statement that only can initialize and increment a numeric counter.

# Getting out of a loop, or the current loop iteration

## return

You can exit a while or for loop by returning out of the enclosing method.

One student wrote this for the TruthGame pickNext method:

```
public int pickNext() {  
    for (int i=1; i<=3; i++) {  
        if (isTruth(i)) {  
            return i;  
        }  
    }  
    return -1;  
}
```

## break

break exits a while or for loop and just moves on to the next statement after the loop.

```
while (true) {  
    String command = scanner.nextLine();  
    if (command.equals("quit")) {  
        break;  
    }  
    ...  
}  
System.out.println("Well, goodbye, then!");
```

## continue

continue jumps back to the top of the loop and re-evaluates the condition. It's useful for skipping the body of the loop, like to ignore blank lines.

```
while (scanner.hasNextLine()) {  
    String line = scanner.nextLine();  
    if (line.equals("")) {  
        continue;  
    }  
    processNonEmptyLine(line);  
}
```

# 4.3

## Looping over Strings

10/14/2022

# Refresher: String Methods!

<code>int length()</code>	Returns the number of characters in a String object.	<code>str.length()</code>
<code>int indexOf(String str)</code> <code>int indexOf(String str, int fromIndex)</code>	Returns the index of the first occurrence of <code>str</code> [starting at <code>fromIndex</code> , if provided].  Returns -1 if not found.	<code>str.indexOf("ing")</code> <code>str.indexOf("ch", 9)</code>
<code>String substring(int from, int to)</code> <code>String substring(int from)</code>	Returns substring beginning at index <code>from</code> and ending at <code>(to - 1)</code> [or <code>length() - 1</code> , if <code>to</code> isn't provided].	<code>str.substring(7, 10)</code> <code>str.substring(3)</code> <code>str.substring(i, i+1)</code>
<code>char charAt(int index)*</code>	Returns the character in the string at <code>index</code> .	<code>str.charAt(2)</code>

\* Note that `charAt` is not part of [AP Computer Science A Java Subset](#)



# String Transformations Using Loops

Many loops over strings are to **transform** a string into another string, e.g., remove spaces, reverse it. This can be approached in multiple ways. Here are two approaches:

**Approach #1: Transform the same String variable repeatedly until you achieve the desired result.**

```
s = "Let us remove all spaces"
s ← "Letus remove all spaces"
s ← "Letusremove all spaces"
s ← "Letusremoveall spaces"
s ← "Letusremoveallspaces"
```

(Remember, Java Strings are immutable, meaning they cannot be modified. When `s` changes, you aren't modifying the same String instance... you are repeatedly changing what the String reference variable `s` points to.)

**Approach #2: Loop over the source String, leaving it unchanged, to build up a new result String.**

```
s = "Let us remove all spaces"
result ← "Let"
result ← "Letus"
result ← "Letusremove"
result ← "Letusremoveall"
result ← "Letusremoveallspaces"
```

Neither approach is always better. It depends on the problem you're solving. You may need to **benchmark** the code both ways to find what works better.

# String loops with while

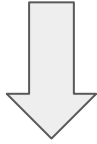
Example: Removes spaces from a String

```
public static String removeSpaces(String s) {  
    int i = s.indexOf(" ");  
  
    // while there is a " " in the string  
    while (i >= 0) {  
        // Remove the " " at index by concatenating  
        // substring up to index and then rest of the string.  
        s = s.substring(0, i) + s.substring(i+1);  
        i = s.indexOf(" ");  
    }  
  
    return s;  
}
```

# When to use `while` vs `for` with strings?

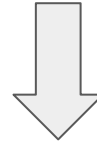
Looking for a certain character or substring?

Don't know how many times the loop needs to run?



`while`

Want to visit every character (e.g. reversing a string, checking if palindrome)?



`for`

# Reverse String Using for Loops

```
class Main {  
    public static String reverseString(String s) {  
        String result = "";  
        for (int i = s.length() - 1; i >= 0; i--) {  
            result += s.charAt(i);  
        }  
        return result;  
    }  
    public static void main(String[] args) {  
        System.out.println(reverseString("Hello world!"));  
    }  
}
```

# Nested Loops

Chapter 4.4

# Rows and Columns

i = 0		j = 0		j = 1		j = 2	
i = 1		j = 0		j = 1		j = 2	
i = 2		j = 0		j = 1		j = 2	
i = 3		j = 0		j = 1		j = 2	
i = 4		j = 0		j = 1		j = 2	

```
for (int i = 0; i < 5; i++) {  
    for(int j = 0; j < 3; j++) {  
        System.out.print(j);  
    }  
    System.out.println();  
}
```

- The **outer loop** iterates through the rows
- The **inner loop** iterates through the columns
- The inner loop runs in its entirety on each iteration of the outer loop

```

class Main {
    public static void main(String[] args) {
        int n = 3;
        int m = 3;
        for (int i=0; i<n; i++) {
            boolean topOrBottom = i == 0 || i == n-1;
            for (int j=0; j<m; j++) {
                char ch = ' ';
                boolean leftOrRight = j == 0 || j == m-1;
                if (topOrBottom && leftOrRight) {
                    ch = '+';
                } else if (topOrBottom) {
                    ch = '-';
                } else if (leftOrRight) {
                    ch = '|';
                }
                System.out.print(ch);
            }
            System.out.println();
        }
    }
}

```

i	j	topOrBottom	leftOrRight	output
0	0	true	true	+
0	1	true	false	-
0	2	true	true	+
1	0	false	true	
1	1	false	false	(space)
1	2	false	true	
2	0	true	true	+
2	1	true	false	-
2	2	true	true	+

# Tracing through primes

```
class Main {  
    public static void main(String[] args) {  
        for (int i=1; i<=100; i++) {  
            boolean p = true;  
            for (int j=2; j<i; j++) {  
                if (i % j == 0) {  
                    p = false;  
                    break;  
                }  
            }  
            if (p) {  
                System.out.println(i);  
            }  
        }  
    }  
}
```

i	j	p (prime)	output
1	2	true	1
2	2	true	2
3	2	true	
3	3	true	3
4	2	false	
5	2	true	
5	3	true	
5	4	true	
5	5	true	5



# Runtime Analysis

Chapter 4.5

# What makes a good algorithm?

From CodeHS:

What makes a good algorithm?

- Correctness
- Easy to understand by someone else
- Efficiency (run time and memory requirements)

**Correctness** refers to whether the algorithm solves the given problem.

**Easy to understand** can be a function of the complexity of the algorithm design, as well as how the code is laid out, use of appropriate variable names, and the use of comments.

**Efficiency** can be looked at in several ways, which we will explore in this section.

# Runtime analysis

How long an algorithm takes to execute (run) is called its **running time** or **runtime**.

**Runtime analysis** is the process of understanding how an algorithm or complete program will perform when it is run.

It includes how long the program takes to run, as well as other factors like how much memory is consumed.

# Timing Execution

`System.currentTimeMillis()` – milliseconds since "Unix Epoch" (January 1, 1970 00:00:00 UTC time zone). This is **wall clock time**... if you adjust the date/time on your computer, the value will change.

`System.nanoTime()` – nanoseconds since an arbitrary point in time (maybe since CPU booted). Independent of "wall clock" ... it will continue increasing even if you futz with computer's date and time.

- Different computers will give you different results, so timing benchmarks are only valid on the same hardware.
- A lot of other stuff can be happening in other processes on a modern computer.
- Different programming languages have different performance characteristics.
- Even the same programming language may behave differently in different environments.

```
> sh -c javac -classpath .:target/dependency/* -d . $(find . -name *.java)
> java -classpath .:target/dependency/* Main
isPalindrome: 0.092981111 s
isPalindromeReversed: 2.723089299 s
isPalindromeReverseBuilder: 0.284532743 s
> █
```

```
StudyMac:~ gary$ javac Main.java
StudyMac:~ gary$ java Main
isPalindrome: 0.03343711 s
isPalindromeReversed: 0.257904175 s
isPalindromeReverseBuilder: 0.065299022 s
StudyMac:~ gary$ █
```

# Statement Execution Count

Getting an absolute number of statements executed is tricky, and some of the things we may want to count are actually just expressions, not full statements.

Here, one way we might do it is to "point" it as follows:

`int i = 3;` counts as 1 point

This part repeats for `i=3, 4, 5, 6`:

`i < 7` counts as 1 point

`System.out.print("*")` counts as 1 point

`i++` counts as 1 point

```
for (int i = 3; i < 7; i++) {  
    System.out.print("*");  
}
```

$1 + 3 * 4 = 13$  "statements" total

There could be different counting methods here. On the AP exam, you'll probably be asked something more like: How many times is `System.out.print` called in this loop? That is, count how many times a **specific** statement is executed.

# Loop Execution Count

You can use a trace table to figure out how many times a loop executes. But, you may be able to shortcut that just looking at it.

With a  $<$  condition, the number of iterations is  $(\text{ending value} - \text{starting value}) = 7 - 3 = 4$  iterations.

```
for (int i = 3; i < 7; i++)  
    System.out.print("*");
```

When a  $\leq$  is involved, the number of iterations is  $(\text{ending value} - \text{starting value} + 1) = 5 - 1 + 1 = 5$  iterations.

```
for (int y = 1; y <= 5; y++)  
{  
    System.out.print("*");  
}
```

Both of these shortcuts assume the increment expression is just adding 1 to the counter.

# Loop Execution Count

For nested loops: The number of times a nested for loop body is executed is the number of times the outer loop runs multiplied by the number of times the inner loop runs (outer loop runs \* inner loop runs).

```
for (int row = 0; row < 5; row++) {  
    for (int col = 0; col < 10; col++) {  
        System.out.print("*");  
    }  
    System.out.println();  
}
```

# Best and worst case

Let's count the loop executions in this implementation of isPalindrome.

The loop execution count depends on the input.

The **best case** is a zero-length string. Next, a one-letter string. Next, the first and last letters do not match, which would be a loop execution count of 0 (but a statement execution count of 3, for the init expression, condition check, and return statement.)

The **worst case** is... if the string is really a palindrome, and we have to check every pair of characters! That would be  $\text{word.length()} / 2$  loop executions.

```
public static boolean isPalindrome(String word) {  
    for (int i = 0, j = word.length() - 1; i < j; i++, j--) {  
        if (word.charAt(i) != word.charAt(j)) {  
            return false;  
        }  
    }  
    return true;  
}
```



# Running time is proportional to...

Let  $N=100$ . (We use capitals for this a lot, for some reason.)

The outer loop executes  $N$  times.

The inner loop executes a variable number of times, but it is bounded by  $N$ .

The loop execution count isn't exactly  $N^2$ ... it's less. But it is proportional to  $N^2$ ; it will grow in concert with  $N^2$ , even if it doesn't follow an exact linear relationship with it.

The running time of this loop is **proportional to  $N^2$** .

An algorithm that runs in  $N^2$  time is said to run in **quadratic time**.

```
class Main {  
    public static void main(String[] args) {  
        for (int i=1; i<=100; i++) {  
            boolean p = true;  
            for (int j=2; j<i; j++) {  
                if (i % j == 0) {  
                    p = false;  
                    break;  
                }  
            }  
            if (p) {  
                System.out.println(i);  
            }  
        }  
    }  
}
```

# Linear time

What is our isPalindrome implementation proportional to?

The loop execution count depends on the input, but it is **bounded** by the length of the input. If the input is length  $N$ , the loop will not execute more than  $N/2$  times.

We can ignore the  $N/2$  factor since we're talking about proportionality.

So, isPalindrome running time is proportional to  $N$ , where  $N$  is the length of the string input.

An algorithm with running time proportional to  $N$  is said to run in **linear time**.

```
public static boolean isPalindrome(String word) {  
    for (int i = 0, j = word.length() - 1; i < j; i++, j--) {  
        if (word.charAt(i) != word.charAt(j)) {  
            return false;  
        }  
    }  
    return true;  
}
```

What about the **average case**?

# Computational Complexity

In [computer science](#), the computational complexity or simply complexity of an [algorithm](#) is the amount of resources required to run it.

**Time complexity** is the amount of time that an algorithm takes to run, but in an abstract sense of how many operations need to be executed, not a quantity of seconds or milliseconds. The usual units of time (seconds, minutes etc.) are not used because they are too dependent on the choice of a specific computer and on the evolution of technology.

**Space complexity** is how much memory an algorithm consumes. A lot of our algorithms so far allocate no memory, or allocate one or two things. This is constant space complexity. Constant space complexity means that the amount of space that your algorithm uses is independent of the input parameters. Linear space complexity means you probably allocated an array of size  $N$ , or for, quadratic,  $N^2$ , etc.

# Big O notation

The time complexity and space complexity of algorithms are often expressed using "Big O notation" – the upper bound (maximum) running time or memory usage.

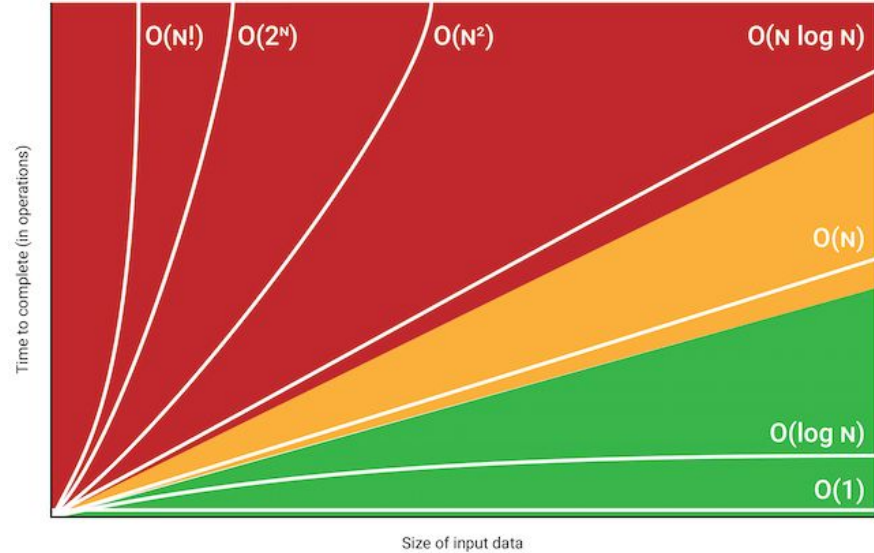
$O(1)$  = Constant time  
(unconnected to length of input)

$O(N)$  = Linear time

$O(N^2)$  = Quadratic time

Other common ones:

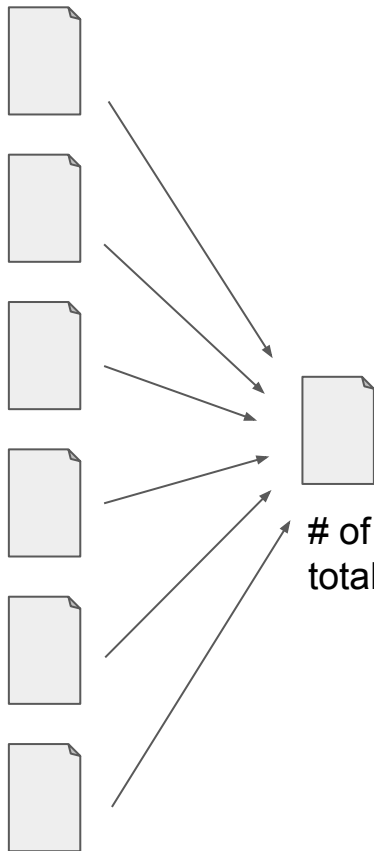
$O(N^3)$ ,  $O(\log N)$ ,  $O(N \log N)$ ,  $O(N!)$ ,  $O(2^N)$



Practice!

## Roll-up / Aggregation

Works of Shakespeare

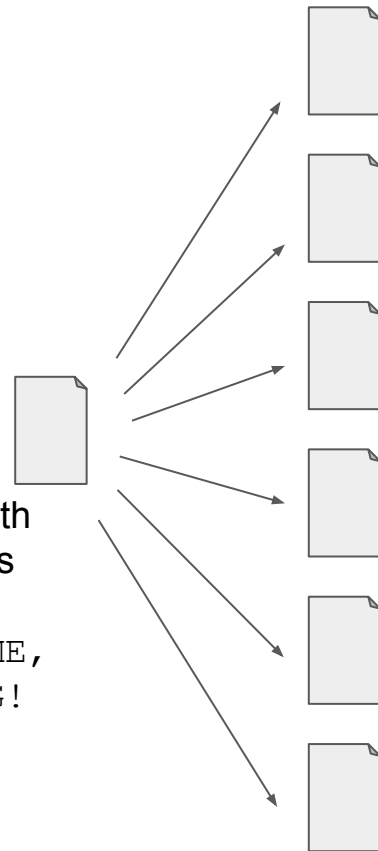


# of words, # of lines  
total and per work

ThankYouMachine

## Mail Merge (Templating)

"Personalized" letters



Template with  
placeholders

Dear \$NAME,  
\$GREETING!