Lecture 4
String in Java

Department of Computer Science
Hofstra University

Lecture Goals

- Describe how Strings are represented in Java Platform
- Perform basic operations with Strings in Java
- Work with the String's built-in methods to manipulate Strings
- Write regular expressions to match patterns and split strings

Motivation Example

There is hereby imposed on the taxable income of every individual (other than a surviving spouse as defined in section 2(a) or the head of a household as defined in section 2(b)) who is not a married individual (as defined in section 7703) a tax determined in accordance with the following table:

Hard to read

26 U.S. Code § 1 – Tax imposed
https://www.law.cornell.edu/uscode/text/26/1

How do we quantify the difference?

If you are single, never lost your spouse, and not the head of a household, you pay taxes according to the following table:

Easy to read

Use **flesch score** to measure of text readability

https://en.wikipedia.org/wiki/Flesch%E2%80%93Kincaid_readability_tests

Measure the Text Readability by Flesch Score

number of words per sentence

number of syllables per word

$$\text{FleschScore} = 206.835 - 1.015\left(\frac{\#\,\text{words}}{\#\,\text{sentences}}\right) - 84.6\left(\frac{\#\,\text{syllables}}{\#\,\text{words}}\right)$$

High score: Few words/sentence and few syllables/word

Low score: Many words/sentence and many syllables/word

longer word makes text harder to read than longer sentence

Document is represented as a big long string. Requires the ability to manipulate Strings!

Score	School level	Notes
100.00-90.00	5th grade	Very easy to read. Easily understood by an ave
90.0–80.0	6th grade	Easy to read. Conversational English for consume
80.0–70.0	7th grade	Fairly easy to read.
70.0–60.0	8th & 9th grade	Plain English. Easily understood by 13- to 15-year-old students.
60.0–50.0	10th to 12th grade	Fairly difficult to read.
50.0–30.0	College	Difficult to read.
30.0–0.0	College graduate	Very difficult to read. Best understood by university graduates.

There is hereby imposed on the taxable income of every individual (other than a surviving spouse as defined in section 2(a) or ... defined in section 2(b)) w... al (as defined in section 7703) a tax determined in accordance with the following table:

FleshScore = 12.6

If you are s..., and not the hea... axes according to the following table:

FleshScore = 65.8

String Basics

```
String text1 = new String("Hello World!");
String text2 = text1;
String text3 = text1.concat("It's a great day.");
String text4 = text1 + "It's a great day.";
String text5 = "Hello World!";
String text6 = "Hello World!";
String text7 = new String("Hello World!");
text7.equals(text1);          // true
text7 == text1;               // false
```

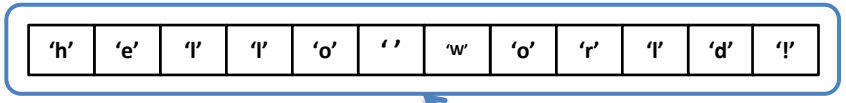
String is an object

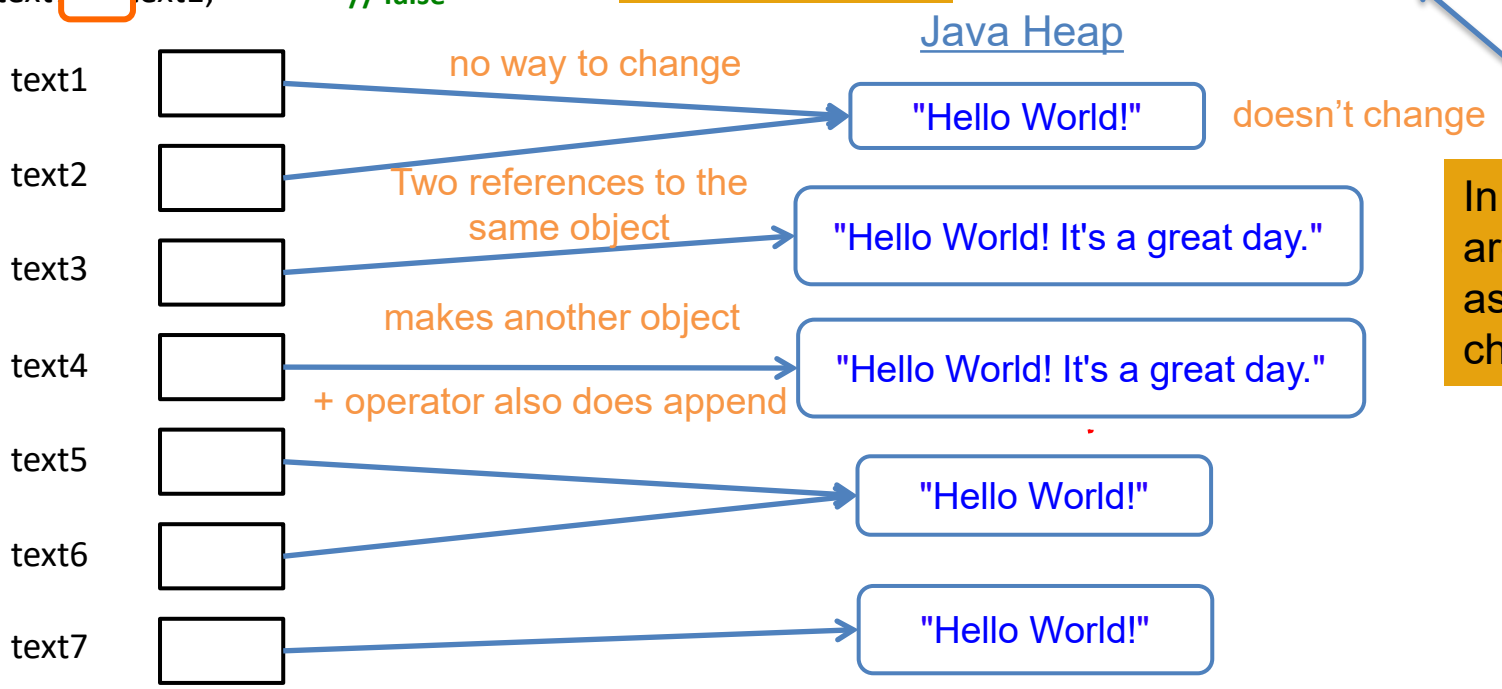
Strings are immutable

String append

String Interning in Java is a process of storing only one copy of each distinct String value ("Hello World!").

Compare string





In heap, Strings are represented as arrays of chars

String Class's Built-in Methods

- Strings can do lots of things:
  - https://docs.oracle.com/javase/10/docs/api/java/lang/String.html
- Let's look at some methods in the context of our problems:
  - `length, charAt, toCharArray, indexOf, split`
- For example, we need to look at words, character by character, to calculate the number of syllables.

does the `letter` appear anywhere in the `word`?

```
public static boolean hasLetter(String word, char letter)
{
        for (int i = 0; i < word.length(); i++) {
                if (word.charAt(i) == letter) {
                        return true;
                }
        }
        return false;
}
```

`charAt(i)` cannot be used to change the String

If no letters match, return false

Loop over the indexes of character array in the string

`length()` returns the number of characters in the String

Get each letter and compare it to the char in question

`charAt(i)` returns the char at index i in the String

Count the number of syllables (Contd.)

```
public static boolean hasLetter(String word, char letter)
{
        for (char c: word.toCharArray()) {
                if (c == letter) {
                        return true;
                }
        }
        return false;
}
```

Same method, using a for-each loop

`toCharArray()` returns the chars in a String, as a char[]

Change this method so that it returns the index where it first finds letter (or -1 if it doesn't find it)?

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

built-in String method `indexOf(String str)` does exactly this, but with a String as argument to be matched.

0

```
String text = "Can you hear me? Hello, hello?
int index = text.indexOf("he");  // index is 8
index = text.indexOf("He");      // index is 17
index = text.indexOf("Help");    // index is -1
```

For dealing with case, check out String methods:
`equalsIgnoreCase, toLowerCase, toUpperCase`

Manipulate String with For-each Loop

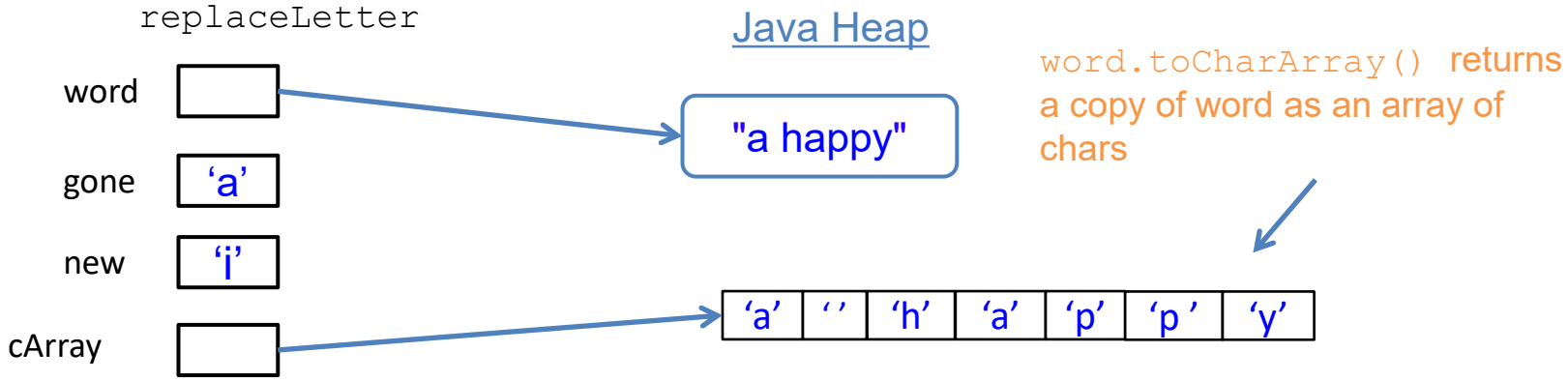
```
public static string replaceLetter(String word, char gone, char new1)
{
        char[] cArray = word.toCharArray();
        for (char c: cArray) {
                if (c == gone) {
                        c = new1;
                }
        }
        return word;
}
```

Does this method successfully return a modified word?

NO, since char c is a copy of each char in cArray, and assigning new1 to c does not change the content of cArray

Let's trace the code with memory model diagram

```
// replaceLetter("a happy", 'a', 'i') -> "i hippy"??
```



Manipulate String with For-each Loop (Contd.)

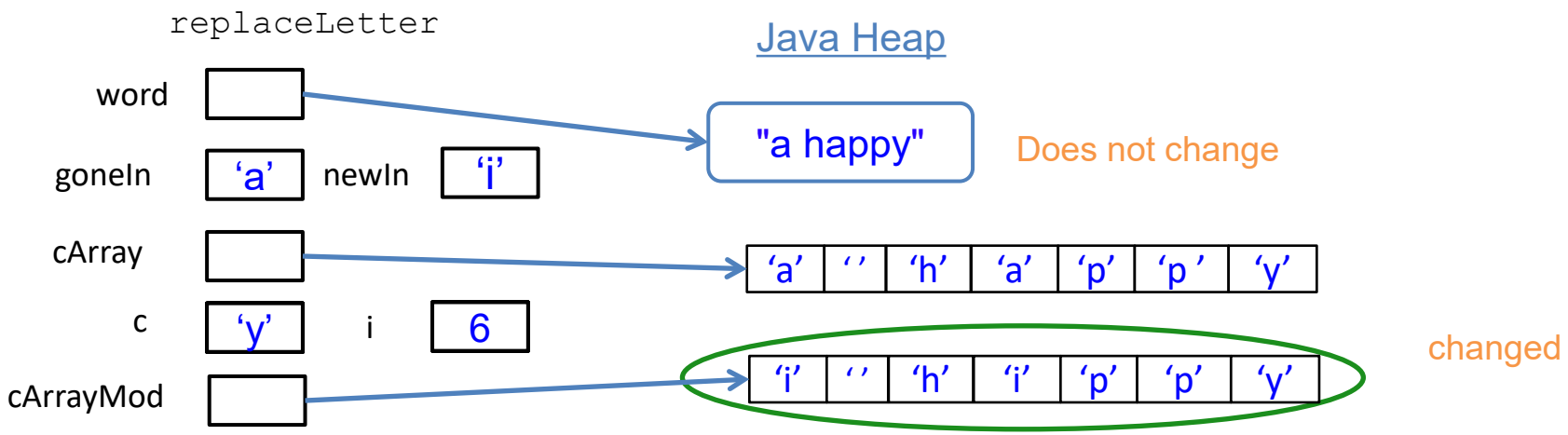
```
public static string replaceLetter(String word, char goneIn, char newIn)
{
        char[] cArray = word.toCharArray();
        char[] cArrayMod = new char[cArray.length];
        int i = 0;
        for (char c: cArray) {
                if (c == goneIn)
                        cArrayMod[i] = newIn;
                else
                        cArrayMod[i] = c;
                i++;
        }
        return new String(cArrayMod);
}
```

Does this method successfully return a modified word?

Attempt #1: NO

Attempt #2: YES

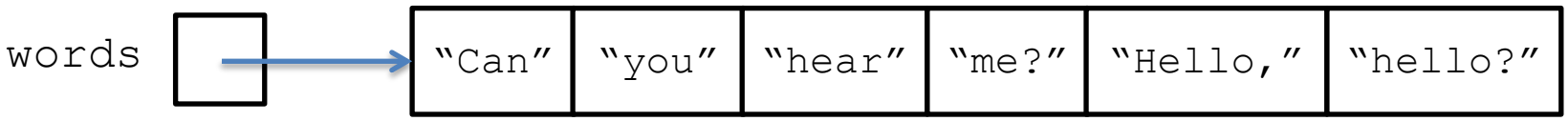
// replaceLetter("a happy", 'a', 'i') -> "i hippy"??



Count number of words in a string

Use String method `split(String regex)` to split apart the String into separate words, where regex stands for regular expression.

```
String text = "Can you hear me? Hello, hello?";
String[] words = text.split(" "); //" “ matches a single space
```

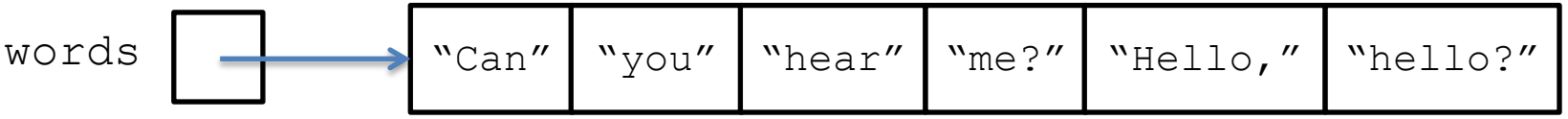


If we have 2 spaces in front of Hello, then the results include an extra space. This is not desirable, as we want the result to contain the 6 words only regardless of how many spaces are in-between words.

```
String text = “Can you hear me?  Hello, hello?";
String[] words = text.split(" "); //" “ matches a single space
```

This can be accomplished by using regex " +“ (space followed by +)

```
String text = “Can you hear me?  Hello, hello?";
String[] words = text.split(" +"); //" +“ matches 1 or more spaces in a row
```



Introduction to Regular Expressions (Regex)

- **Regular expression** ("regex"): describes a pattern of text
  - can test whether a string matches the expr's pattern
  - can use a regex to search/replace characters in a string
- Regular expressions occur in many places:
  - text editors (TextPad) allow regexes in search/replace
  - Unix/Linux/Mac shell commands (`grep`, `sed`, `find`, etc.)
  - languages: Java, JavaScript

`.match(`***`regexp`***`)`	returns first match for this string against the given regular expression; if global /g flag is used, returns array of all matches
`.replace(`***`regexp, text`***`)`	replaces first occurrence of the regular expression with the given text; if global /g flag is used, replaces all occurrences
`.search(`***`regexp`***`)`	returns first index where the given regular expression occurs
`.split(`***`delimiter[, limit]`***`)`	breaks apart a string into an array of strings using the given regular as the delimiter; returns the array of tokens

Java String regexp methods

Wildcards and anchors

`.` (a dot) matches any character except newline `\n`

- `.oo.y` matches "Doocy", "goofy", "LooPy", ...
- use `\.` to match a literal dot `.` character

`^` matches the beginning of a string or line; `$` the end

- `^hello` matches: "hello world", but not "world hello“
- `world$` matches: "hello world", but not "world hello“
- `^hello$` matches: "hello" (only if "hello" is the entire string), but not "hello world" or "world hello“

`\<` demands that pattern is the beginning of a *word*
`\>` demands that pattern is the end of a word

- `\<cat` matches: "cat" in "catfish" or "a cat", but not "concatenate“
- `cat\>` matches: "cat" in "black cat" or "concat", but not "category“
- \<cat\> matches: "cat" as a standalone word, but not if it is part of another word such as: "category“, "concatenate"

Special characters

| means OR

- `abc|def|g` matches `"abc"`, `"def"`, or `"g"`
- precedence: `^Subject|Date:` vs. `^(Subject|Date):`

() are for grouping

- `(Homer|Marge) Simpson` matches `"Homer Simpson"` or `"Marge Simpson"`

\ starts an escape sequence, to treat the letter after it as a literal with no special meaning

- many characters must be escaped: / \ $. [ ] ( ) ^ * + ?
- `\.\\n` matches the string .\n

Quantifiers: * + ? {min,max}

`*` means 0 or more occurrences
- `abc*` matches "ab", "abc", "abcc", "abccc", ...
- `a(bc)*` matches "a", "abc", "abcbc", "abcbcbc", ...
- `a.*a` matches "aa", "aba", "a8qa", "a!?_a", ...

`+` means 1 or more occurrences
- `a(bc)+` matches "abc", "abcbc", "abcbcbc", ...
- `Goo+gle` matches "Google", "Gooogle", "Goooogle", ...

`?` means 0 or 1 occurrences
- `Martina?` matches lines with "Martin" or "Martina"
- `Dan(iel)?` matches lines with "Dan" or "Daniel"

`{min,max}` means between ***min*** and ***max*** occurrences (***min*** or ***max*** may be omitted to specify no lower or upper bound)
- `a(bc){2,4}` matches "abcbc", "abcbcbc", or "abcbcbcbc"
- `{2,}` means 2 or more repetitions
- `{,6}` means 0 up to 6 repetitions
- `{3}` means exactly 3 repetitions

Character sets

`[ ]` group characters into a *character set*; will match any single character from the set

- `[bcd]art` matches "bart", "cart", and "dart"; equivalent to `(b|c|d)art` but shorter

Inside `[ ]`, most modifier keys act as regular characters

- `what[.!*?]*` matches "what", "what.", "what!", "what?**!", since these characters inside a character set `.!*?` lose their special meaning and are treated as literal characters

Character ranges

- an initial `^` inside a character set negates it
  - `[^abcd]` matches any character but a, b, c, or d
  - `[^a-cz]` matches any character that is not between a-c and not z

- inside a character set, specify a range of chars with `-`
  - `[a-z]` matches any lowercase letter between a and z
  - `[a-zA-Z0-9]` matches any letter or digit
  - `[a-z]?` matches zero or one lowercase letter, incl. the empty string
  - `[a-z]*` matches zero or more lowercase letters, incl. the empty string
  - `[a-z]+` matches one or more lowercase letters.

- inside a character set, `-` must be escaped to be matched, unless it appears at the beginning or end:
  - `[0-9]` is equivalent to `\d`, and matches any single digit 0 through 9
  - `[0\-9]` matches a single digit 0 or 9, or a literal hyphen -
  - `[-+]?[0-9]+` matches signed or unsigned integers (e.g., 8, -8, +23). (`[\-+]?[0-9]+` is equivalent, but the escape `\` is unnecessary since – appears at the beginning)

- Example: match a US phone number with optional spaces or dashes as separators (e.g., 2066852181, 206 685 2181, 206-685-2181)
  - `\d{3}[ -]?\d{3}[ -]?\d{4}` or equivalently, `[0-9]{3}[ -]?[0-9]{3}[ -]?[0-9]{4}`
  - `\d{3}, [0-9]{3}:` matches exactly 3 digits (0 through 9)
  - `[ -]?:` matches an optional space or hyphen between digit groups

Built-in character ranges

- `\b` word boundary (e.g. spaces between words)
- `\B` non-word boundary
  - \bcat\b (same as \<cat\>) matches: "cat" in "a black cat“, but not "cat" in "certificate“
    - \b is a general word boundary that matches both the start and end of a word. \< and \> are more specific: \< matches only the start of a word, \> matches only the end of a word
  - \Bcat\B matches: "cat" in "certificate“, but not "cat" in "a black cat"
- `\d` any digit; equivalent to `[0-9]`
- `\D` any non-digit; equivalent to `[^0-9]`
- `\s` any whitespace character; `[ \f\n\r\t\v...]`
  - Space , Tab (\t), Newline (\n), Carriage return (\r), Vertical tab (\v), Form feed (\f)…
- `\S` any non-whitespace character
- `\w` any word character; `[A-Za-z0-9_]`
- `\W` any non-word character
  - `\w+\s+\w+` matches two space-separated words

Create More Complicated Regex

```
public class Document {
        private String text; // The text of the whole document
        protected List<String> getTokens(String pattern)
}
```

returns a List of "tokens"

regex defining the "tokens"

Document object, d, contains text "Hello hello?", with 2 spaces between Hello and hello

```
d.getTokens(" +");
-> ["  "]
```

Matches 1 or more spaces

Repetition

Assume you have a Document object, d, whose text is "Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("it");
-> ["it", "it"]
```

Matches string “it”

Assume you have a Document object, d, whose text is "Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("it+");
-> ["itt", "it"]
```

Matches string i followed by 1 or more t’s

Create More Complicated Regex (Contd.)

```
public class Document {
        private String text; // The text of the whole document
        protected List<String> getTokens(String pattern)
}
```

Assume you have a Document object, d, whose text is "Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("i(t+)");
-> ["itt", "it"]
```

Same as "it+"; Use parens to group if you are unsure of grouping

Assume you have a Document object, d, whose text is "Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("it*");
-> ["itt", "i", "i", "it", "i"]
```

Matches string i followed by 0 or more t's

Assume you have a Document object, d, whose text is "Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("it|st");
-> ["it", "st", "it"]
```

| means OR

Alternation

Create More Complicated Regex (Contd.)

```
public class Document {
        private String text; // The text of the whole document
        protected List<String> getTokens(String pattern)
}
```

Assume you have a Document object, d, whose text is "Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("[123]");
-> ["1", "2" , "3", "3"]
```

[ ] mean match "anything in the set"

Character classes

```
d.getTokens("[1-3]");
-> ["1", "2" , "3", "3"]
```

- indicates a range
(any character between 1 and 3)

Assume you have a Document object, d, whose text is "Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("[a-f]");
-> ["a", "a", "e", "a", "a"]
```

- indicates a range
(any character between a and f)

Assume you have a Document object, d, whose text is "Splitting a string, it's as easy as 1 2 33! Right?"

```
d.getTokens("[^a-z123 ]");
-> ["S", ",", "'", "!", "R", "?"]
```

^ indicates NOT any characters in this set (including the empty space)

Negation

Quiz

```
public class Document {
        private String text; // The text of the whole document
        protected List<String> getTokens(String pattern)
}
```

Assume you have a Document object, d, whose text is "`Splitting a string, it's as easy as 1 2 33! Right?`"

```
d.getTokens("__________");
-> ["1", "2" , "33"]
```

Which of the following regular expressions can you insert in the blank to get the output shown?

A. "[1233]"	✗	-> ["1", "2", "3", "3"]	Matches “1” or “2” or “3”, same as [123]
B. "[1,2,33]"	✗	-> [",", "1", "2", "3", "3"]	Matches “1” or “,” or “2” or “3”
C. "[0-9]+"	✓	-> ["1", "2" , "33"]	Matches any non-negative integer. More flexible and useful than D.
D. "1|2|33"	✓	-> ["1", "2" , "33"]	Matches any one of “1” or “2” or “33”

Use Regex to Calculate Flesch Score

```
public class Document {
        private String text; // The text of the whole document
        protected List<String> getTokens(String pattern)
        public abstract int getNumWords();
        public abstract int getNumSentences();
```

given helper method

```
public class BasicDocument extends Document {
        @Override
        public int getNumWords() {
                List<String> tokens = getTokens("_____________");
                return tokens.size();
        }
        @Override
        public int getNumSentences()
        {
                List<String> tokens = getTokens("_____________");
                return tokens.size();
        }
```

Need a regex that matches "any word"

What constitutes a word?
"Any contiguous sequence of alphabetic characters"

What constitutes a sentence?
"A sequence of any characters ending with end of sentence punctuation (. ! ?)"

https://www.tutorialspoint.com/java/java_regular_expressions.htm

Regex Exercises

- `^re*ed$`
  - Matches strings that start with "r", with e repeated 0 or more times, and end with "ed" (like "reed" or "reeed" or "reeeeeeeed")
- `^(re)*ed$`
  - ed, reed, rereed, rererereered
- `^[re]*ed$`
  - ed, eed, red, rrrred, eerreerred, rerereed
- `^[re]+ed$`
  - eed, red, rrrred, eerreerred, rerereed, but NOT ed
- `^re{2}ed$`
  - reeed
- `^(re){2}ed$`
  - rereed
- `^re\wed$`
  - \w Matches any single word character (letter, digit, or underscore)
  - Matches "re" followed by exactly one word character, followed by "ed" (like rexed, re1ed, re_ed, reAed)
- `^re\w+ed$`
  - Matches "re" followed by one or more word characters, followed by "ed" (like received, renewed)
- `^re.ed$`
  - . Matches any single character (except newline)
  - Matches "re" followed by exactly one single character, followed by "ed" (like rexed, re-ed, re ed (including a space), re3ed, re.ed (matching a literal period)

Quiz: IPv4 Address

- Which regex matches a valid IPv4 address? An IPv4 address consists of four 8-bit segments (octets) separated by periods, such as 192.168.0.1. Each octet can range from 0 to 255.
  - A. \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
  - B. (\d{1,3}\.){3}\d{1,3}
  - C. (25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?)(\.(25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?)){3}
  - D. [0-255]\.[0-255]\.[0-255]\.[0-255]
- ANS: C

Quiz: IPv4 Address

- Correct choice C: (25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?)(\.(25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?)){3}
- This regex pattern consists of two main parts:
  1. (25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?): This part matches a single octet (0-255) of an IPv4 address.
  2. (\.(25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?)){3}: This part matches the remaining three octets, each preceded by a dot.
- **Matching a Single Octet**
  - The first part (25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?) matches numbers from 0 to 255:
  - 25[0-5]: Matches numbers from 250 to 255
  - 2[0-4][0-9]: Matches numbers from 200 to 249
  - [1]?[0-9][0-9]?: Matches numbers from 0 to 199. ? is a quantifier that specifies that the preceding element (in this case, "1") can appear zero or one time. Essentially, it makes the presence of "1" optional.
- **Matching the Full IP Address**
  - The second part (\.(25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?)){3} repeats the octet pattern 3 more times, each preceded by a dot. The backslash (\) is used to escape the dot because, in regex, a dot normally matches any character except a newline.

Quiz: IPv4 Address

- Wrong choice A, B: \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}, or (\d{1,3}\.){3}\d{1,3}
- For \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}:
  - \d{1,3}: Matches between one and three digits. This pattern is repeated four times, once for each segment of an IPv4 address.
    - \d: Matches any single digit from 0 to 9.
    - {1,3}: Specifies that the preceding element (a digit) must occur at least once and at most three times.
  - \.: Matches the literal dot character.
  - This regex will match strings that look like IPv4 addresses, such as "192.168.0.1", "10.0.0.255", or "127.0.0.1". However, it does not validate whether each segment is within the valid range for an IPv4 address (0 to 255), so it will match “292.999.0.1".
- Choices A and B are the same, since (\d{1,3}\.){3} is the same as \d{1,3}\.\d{1,3}\.\d{1,3}\.
- Wrong choice D: [0-255]\.[0-255]\.[0-255]\.[0-255]
  - The pattern [0-255] matches any single character that is either “0”, or “1”, or “2”, or “5”. It is equivalent to [0125]. This does not correctly represent the range of numbers from 0 to 255.

Quiz: time in 24-hour format (HH:MM)

- Which regex can be used to match a valid time in 24-hour format (HH:MM)?
  - A. \d\d:\d\d
  - B. [0-2]\d:[0-5]\d
  - C. (\d|1[0-9]|2[0-3]):[0-5]\d
  - D. [0-9]{2}:[0-9]{2}
- ANS: C

Quiz: time in 24-hour format (HH:MM)

- Correct choice C: (\d|1[0-9]|2[0-3]):[0-5]\d
- (\d|1[0-9]|2[0-3]): Matches the hour part of the time.
  - \d: Matches any single digit from 0 to 9, which would cover hours "0" to "9".
  - 1[0-9]: Matches hours from "10" to "19".
  - 2[0-3]: Matches hours from "20" to "23". ([20-23] is incorrect, since it matches 203, 213, 223)
- : Matches the colon character that separates the hours and minutes.
- [0-5]\d: Matches the minutes part of the time.
  - [0-5]: Matches any digit from 0 to 5, representing the tens place of minutes.
  - \d: Matches any single digit from 0 to 9, representing the units place of minutes. Equivalent to [0-9].
- This regex pattern effectively captures valid hour and minute combinations in a 24-hour time format, such as "3:15", "12:45", and "23:59". However, it allows for single-digit hours without a leading zero (e.g., "3:15" instead of "03:15").
- To ensure that hours are always two digits like "03:15": [01][0-9]|2[0-3]):[0-5]\d
- To allow the hour (HH) be either one or two digits (e.g., 3:45 or 03:45): [0-9]|[01][0-9]|2[0-3]):([0-5] \d

Quiz: time in 24-hour format (HH:MM)

- Wrong choices A, D: \d\d:\d\d, or [0-9]{2}:[0-9]{2}
  - Both are the same, matches strings like "12:34", "99:99", or "00:00"
- Wrong choice B: [0-2]\d:[0-5]\d
  - [0-2]\d: This part matches the hour component of the time.
    - [0-2]: Matches any single digit from 0 to 2, representing the tens place of the hour.
    - \d: Matches any single digit from 0 to 9, representing the units place of the hour. Combined with [0-2], this allows for hour values from "00" to "29". However, this pattern is slightly incorrect since it allows hours like "25" to "29", which are not valid.
  - : Matches the colon character that separates hours from minutes.
  - [0-5]\d: This part matches the minute component of the time.
    - [0-5]: Matches any digit from 0 to 5, representing the tens place of the minutes.
    - \d: ensures that minute values range from "00" to "59".
  - While this regex pattern captures many valid times, it incorrectly allows some invalid hour values (like "25:00").

Quiz

- 1) Dates in the format MM/DD/YY or MM/DD/YYY.
- (0[1-9]|1[0-2])\/(0[1-9]|[12]\d|3[01])\/(\d{2}(\d{2})?
- - The first group ensures a valid month (01–12).
- - The second group ensures a valid day (01–31).
- - The third captures the year in either two or four digits.
- 2) The first alphabetic word (upper or lower case) at the start of the string.
- ^[A-Za-z]+
- - The caret (^) anchors the match at the start, and the character class covers letters only.
- 3) Any price in the form $3.45, $23.32, or $400.
- \$\d+(\.\d{2})?
- - Escapes the dollar sign.
- - Matches one or more digits followed by an optional decimal part with exactly two digits.
- - Does each of the following match? $1.11, $1.1, $1., $1, 1, 1.11
- 4) Words that start with a vowel (a, e, i, o, u), may have any number of letters (a–z) after, and end with a consonant.
- \b[aeiouAEIOU]([a-zA-Z]*[^aeiouAEIOU])\b
- - Uses word boundaries (\b).
- - Ensures the first character is a vowel and the last character is a consonant.
- 5) Words that start with an uppercase letter followed by at least one lowercase letter (e.g., “Apple” or “Banana”).
- \b[A-Z][a-z]+\b
- - The pattern forces an uppercase letter first and one or more lowercase letters afterward, bounded by word boundaries.
- 6) Either “cat” and “cats”.
- cats?
- - The “s” is marked as optional by the quantifier “?”.
- 7) “abc” followed by zero or more digits.
- abc\d*
- - Matches “abc” exactly, followed by any number (including zero) of digits.
- 8) Either “gray” and “grey”.
- gr[ae]y
- - Brackets allow either “a” or “e” in the third character position.
- 9) “br” followed by any single character (except newline) and then “3”.
- br.3
- - The dot (.) matches any character (other than newline).
- 10) The literal string “t.forward”.
- t\.forward
- - The period is escaped with a backslash to treat it literally.

Quiz

- 11) Exactly eight word characters (letters, digits, or underscores).
- \w{8}
- - Uses the shorthand "\w" for word characters and the quantifier "{8}" for exactly eight occurrences.
- 12) One or more lowercase letters followed by a space and then two to four digits.
- [a-z]+\s\d{2,4}
- - "[a-z]+" ensures one or more lowercase letters; "\s" matches a space, and "\d{2,4}" matches two to four digits.
- 13) A regex that captures numbers composed of 5 to 7 digits (e.g., "48105", "103028", "1234567").
- \d{5,7}
- - "\d{5,7}" directly restricts the string to 5–7 digit characters.
- 14) Strings having two digits followed by a period and then exactly four letters (from a to z).
- \d{2}\.[a-z]{4}
- 15) Valid email addresses where the local part is alphanumeric (allowing dots, underscores, %, +, and −), followed by "@", and then a domain that contains letters, numbers, hyphens, and at least one dot.
- [A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}
- 16) US phone number with optional spaces or dashes as separators (e.g., 2066852181, 206 685 2181, 206-685-2181)
- \d{3}[-]?\d{3}[-]?\d{4}
- 17) US phone numbers in either the format "333-232-3403" or "(333) 232 3403".
- (\(\d{3}\)\s|\d{3}-)\d{3}[-\s]\d{4}
- - Either an area code in parentheses followed by a space or three digits followed by a hyphen. (You can also write a single space instead of \s, but \s has better readability, even though it matches more than the white space.)
- - Then matches three digits, a separator (hyphen or space), and four digits.
- 18) US phone numbers in various formats, including "123-456-7890", "(123) 456-7890", "123.456.7890", and "1234567890".
- (\(\d{3}\)|\d{3})[-.\s]?\d{3}[-.\s]?\d{4}
- - Uses an optional set of delimiters (hyphen, dot, or space) and allows the area code to be either parenthesized or plain.
- 19) Hexadecimal color codes such as "#FFA07A" or "#ffa07a" where "#" is mandatory and followed by exactly six hexadecimal digits.
- #[A-Fa-f0-9]{6}
- - The pattern starts with "#" and then exactly six characters from a–f, A–F, or 0–9.

Quiz

- 1) Any string that starts with "cat" and ends with "dog".
- ^cat.*dog$
- 2) All dates in the format MM/DD/YYYY or MM-DD-YYYY.
- \d{2}[/-]\d{2}[/-]\d{4}
- 3) Words starting with a vowel and ending with a consonant.
- \b[aeiouAEIOU][a-zA-Z]*[^aeiouAEIOU]\b
- 4) Strings containing only lowercase letters and digits, with a length of 5–10 characters.
- [a-z0-9]{5,10}
- 5) All words in a string that are exactly 4 characters long.
- \b\w{4}\b
- 6) Strings that contain the substring "abc" followed by zero or more digits.
- abc\d*
- 7) Binary strings containing at least three consecutive 1s.
- (0|1)*111(0|1)*
- 8) Numbers with 5–7 digits (e.g., 12345, 1234567).
- \d{5,7}
- 9) Dates in the format YYYY-MM-DD.
- \d{4}-\d{2}-\d{2}
- 10) All occurrences of the word "cat" or "dog" as whole words.
- \b(cat|dog)\b
- 11) Strings where the number of "a"s is greater than or equal to 3.
- [^a]*a[^a]*a[^a]*a.*
  - The regex looks for: Zero or more non-"a" characters ([^a]*). A literal "a". Zero or more non-"a" characters ([^a]*) followed by another "a". Zero or more non-"a" characters ([^a]*) followed by a third "a". Any remaining characters in the string (.*).