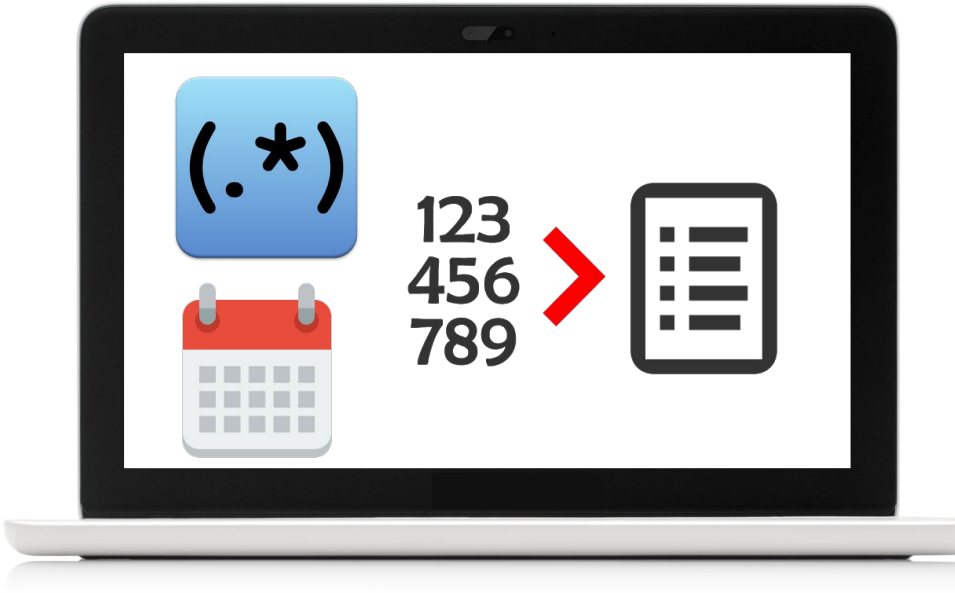
# Regex, Enums, Dates

Please go through material for these topics. Complete the reading, exercises, and any videos linked. If the instructions ask to turn in any exercises, please do so through slack to your instructor.




Cognixia™

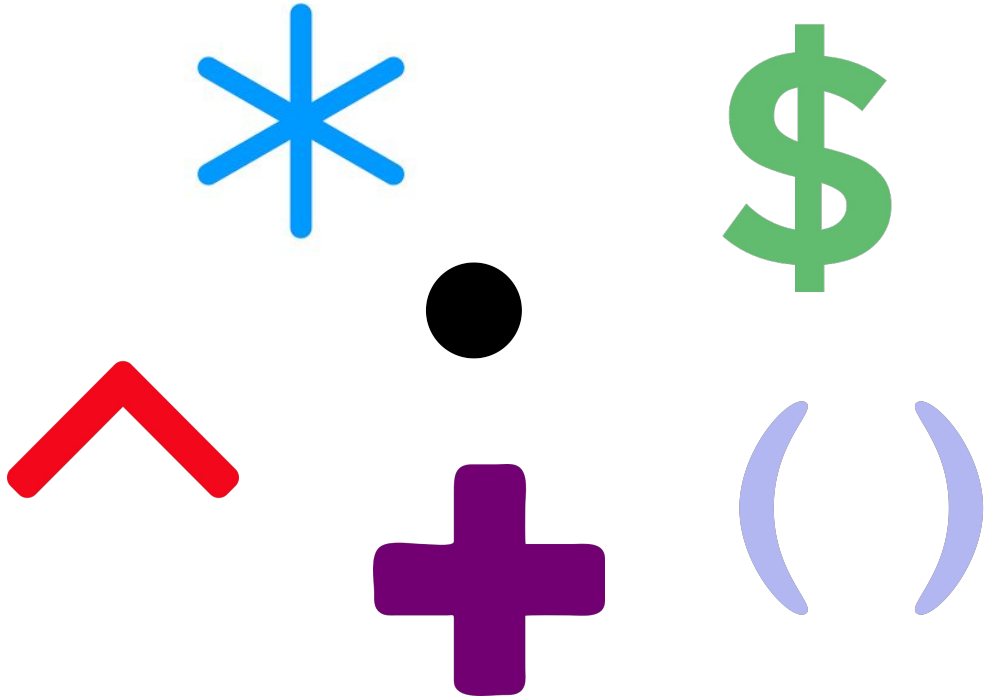
# Outline

1. Regex
    - What is Regex?
    - Regex Rules
    - Exercise
    - Regex in Java
2. Enums
    - Video on Enums
    - Enums in Java
3. Dates
    - Date from java.util
    - New Java 8 Dates
    - Exercise
4. Open Book Quiz

# Instructions

- ➔ Please review all sections on the topics listed in the outline
  - ◆ Read through material, watch accompanying videos, run coding examples, etc.
- ➔ **Exercise are optional** and you are **NOT required to turn them in** unless otherwise stated
  - ◆ It is ***recommended you try them*** regardless to help understand the topics better
- ➔ The **open note quiz** at the end is **REQUIRED**
  - ◆ Have it ***turned in by the start of class the next day***
  - ◆ Ask your instructor if you have any questions regarding this
- ➔ Once you have turned in the quiz, feel free to leave for the day

# Regular Expressions



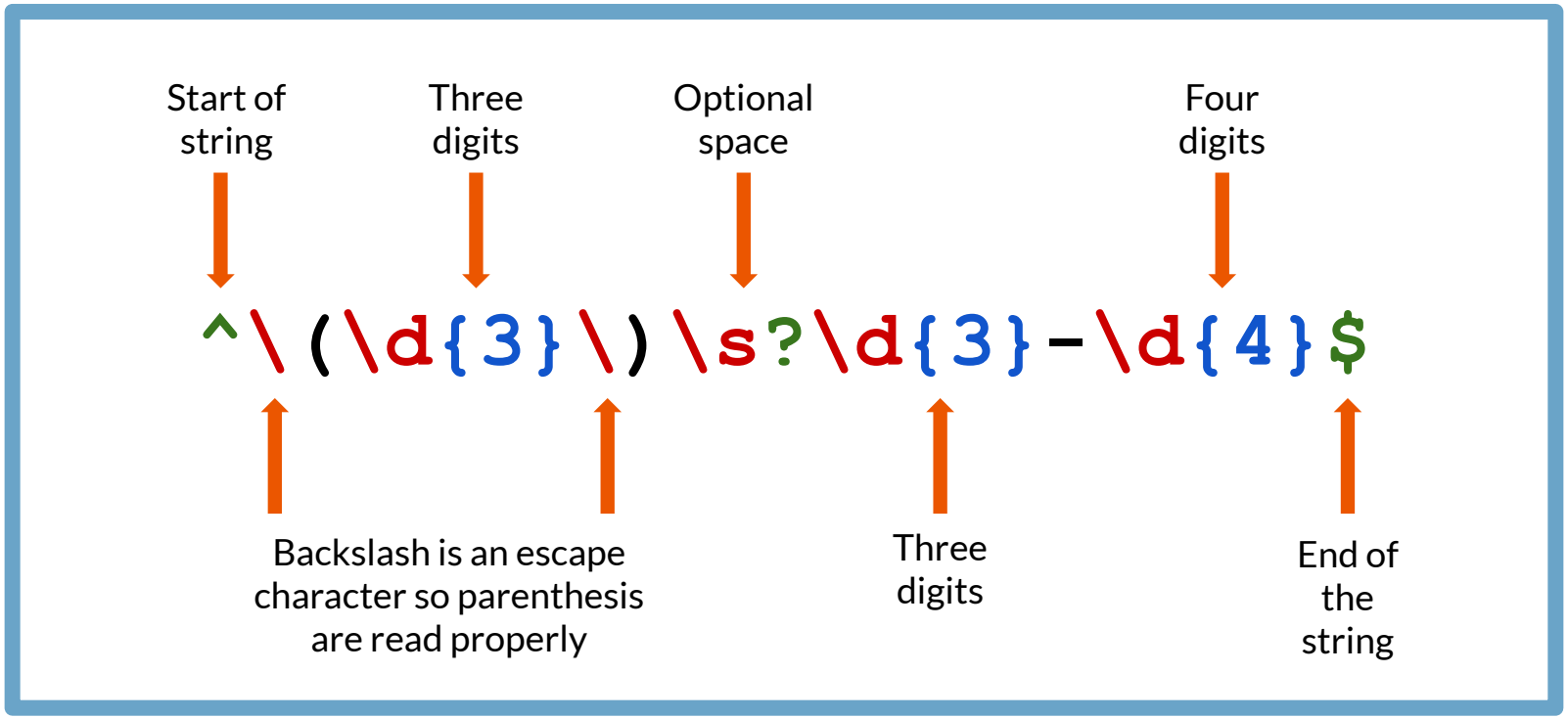
# REGEX: Pattern Matching

A **regular expression** is a string that describes a search pattern. They can be used to find certain substring patterns in a string or used to validate user input.

```
^\(\d{3}\)\s?\d{3}-\d{4}$
```

```
(012)345-6789
```

Matches a phone number of pattern:
(XXX) XXX-XXXX


Start of string
Three digits
Optional space
Four digits
^\(\d{3}\)\s?\d{3}-\d{4}$
Backslash is an escape character so parenthesis are read properly
Three digits
End of the string

(XXX) XXX-XXXX

# Pattern Matching Rules

- ➔ Patterns are created using a ***reserved set of symbols*** to indicate when a characters show up in the matching string
- ➔ **Regex doesn't need to be memorized**
- ➔ Always ***look back to references*** when writing new regexes
- ➔ Next slides detail some of the more common regex rules
- ➔ References if looking to learn more on regex:
  - ◆ https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285
  - ◆ https://medium.com/tech-tajawal/regular-expressions-the-last-guide-6800283ac034

| ^ | First use: denotes what the ***start of the string*** should look like. |
|---|---|
| | Second use: ***negation of a pattern*** (matches if character sequence specified is NOT present) when paired with square brackets [ ] |

Pattern: `^Hello`

| Matches | Doesn't Match |
|---|---|
| "Hello" | "hello" |
| "Hello World" | "Hi Hello" |
| "Hello123" | "  Hello" |

Pattern: `[^a]`

| Matches | Doesn't Match |
|---|---|
| "tree" | "apple" |
| "Hello" | "sand" |
| "Apple" | "Java" |

| $ | Denotes the ***end of a string***, will define what the end of a regex pattern will look like. |
|---|---|

Pattern: `Hello$`

| Matches | Doesn't Match |
|---|---|
| "Hello" | "hello" |
| "Welcome and Hello" | "Hi hello" |
| "123Hello" | "  hello" |

# Regex Without ^ and $

- ^ and $ in a the beginning and end of a regex makes sure to ***match for the whole string***
- If not placed, will look for ***substring matches***
- Example on right will match for different strings depending on if they are placed on the *Hello* or not

Pattern: `^Hello$`

| Matches | Doesn't Match |
|---|---|
| "Hello" | "Hello World" |

Pattern: `Hello`

| Matches | Doesn't Match |
|---|---|
| "Hello" | "hello" |
| "Hi Hello" | "Hi hello" |

| . | A single period will ***represent any single character***. This character can be alphanumeric or a symbol. |
|---|---|

Pattern: `^b.t$`

| Matches | Doesn't Match |
|---|---|
| "bit" | "boot" |
| "b5t" | "b123t" |
| "b@t" | "baan" |

| * | Denotes ***zero or more*** occurences of a single character or a set of characters. |
|---|---|

Pattern: `^ba*b$`

| Matches |
|---|
| "bb" |
| "bab" |
| "baaab" |

Pattern: `^b.*b$`

| Matches |
|---|
| "bacb" |
| "b1@cb" |
| "baaacccccb" |

Pattern: `^b(ac)*b$`

| Matches |
|---|
| "bacb" |
| "bacacacacb" |
| "bb" |

| + | Denotes ***one or more*** occurences of a single character or a set of characters. |
|---|---|

Pattern: `^ba+b$`

| Matches | Doesn't Match |
|---|---|
| "bab" | "bb" |
| "baaab" | "babab" |
| "baaaaaaaaab" | "bbbb" |

Pattern: `^b(ac)+b$`

| Matches | Doesn't Match |
|---|---|
| "bacb" | "bb" |
| "bacacb" | "babacb" |
| "bacacacb" | "bcb" |

| ? | Denotes ***zero or one*** occurences of a single character or a set of characters. |
|---|---|

Pattern: `^ba?b$`

| Matches | Doesn't Match |
|---|---|
| "bab" | "baab" |
| "bb" | "baaaabb" |

Pattern: `^b(ac)?b$`

| Matches | Doesn't Match |
|---|---|
| "bacb" | "bacacb" |
| "bb" | "bacacacbacb" |

| \ | Used as an ***escape character*** so that certain reserved symbols can be used literally in pattern matching. |
|---|---|

Pattern: `^ab\*a$`

| Matches | Doesn't Match |
|---|---|
| "ab*a" | "ab/a" |

Pattern: `^\(hello\)$`

| Matches | Doesn't Match |
|---|---|
| "(hello)" | "hello" |

Pattern: `^\^hello$`

| Matches | Doesn't Match |
|---|---|
| "^hello" | "^^hello" |


Specifies a ***single number or a range*** that a character or set of characters can appear in the string.

Pattern: `^ba{2}b$`

| Matches | Doesn't Match |
|---|---|
| "baab" | "bab" |

Pattern: `^ba{5}b$`

| Matches | Doesn't Match |
|---|---|
| "baaaaab" | "bab" |

Pattern: `^b(ac){3,5}b$`

| Matches | Doesn't Match |
|---|---|
| "bacacacb" | "bacacb" |
| "bacacacacb" | "bacb" |
| "bacacacacacb" | "bb" |

( | )

***List of characters or sets of characters*** listed can be used to fit the regex pattern.

Pattern: `^b(a|i|e)t$`

| Matches | Doesn't Match |
|---|---|
| "bat" | "bot" |
| "bit" | "bt" |
| "bet" | "baat" |

Pattern: `^(ch|m|sk)at$`

| Matches | Doesn't Match |
|---|---|
| "chat" | "brat" |
| "mat" | "drat" |
| "skat" | "at" |

| [ ] | ***List of characters*** can be used to fit the regex pattern. Can't be used for options that include two or more characters like last example. |
|---|---|

Pattern: `^b[aie]t$`

| Matches | Doesn't Match |
|---|---|
| "bat" | "bot" |
| "bit" | "bt" |
| "bet" | "baat" |

Pattern: `^[cms]at$`

| Matches | Doesn't Match |
|---|---|
| "cat" | "skat" |
| "mat" | "drat" |
| "sat" | "at" |

| [0-9] | ***Digits from zero to nine***. Can be shortened to other variations like [2-5] so you only use digits within that range. |
|---|---|

Pattern: `^[0-9][0-9]$`

| Matches | Doesn't Match |
|---|---|
| "00" | "7" |
| "87" | "123" |
| "24" | "0" |

| [a-z ] | ***Lowercase letters from a to z***. Can be shortened to other variations like [a-d] so you only use letters within that range. |
|---|---|

Pattern: `^[a-z]+$`

| Matches | Doesn't Match |
|---|---|
| "abc" | "hello7abc" |
| "hello" | "hEllo" |
| "xyz" | "1234" |

| [a-z A-Z] | ***Lowercase & uppercase letters from a to z***. Will take into account casing if looking to just match letters. |
|---|---|

Pattern: `^[a-zA-Z]+$`

| Matches | Doesn't Match |
|---|---|
| `"aBC"` | `"ABC7abc"` |
| `"HELLO world"` | `"Hello$World"` |
| `"xyzABC"` | `"1234"` |

| | |
|---|---|
| /w | ***A word character***. This includes numbers, letters, and underscores. |
| /d | ***A digit***. Any number from 0 to 9, think back to [0-9] pattern matcher. |
| /s | ***A whitespace character***. This includes single spaces, new line characters, and tabs. |

# Reg (Ex) Exercise

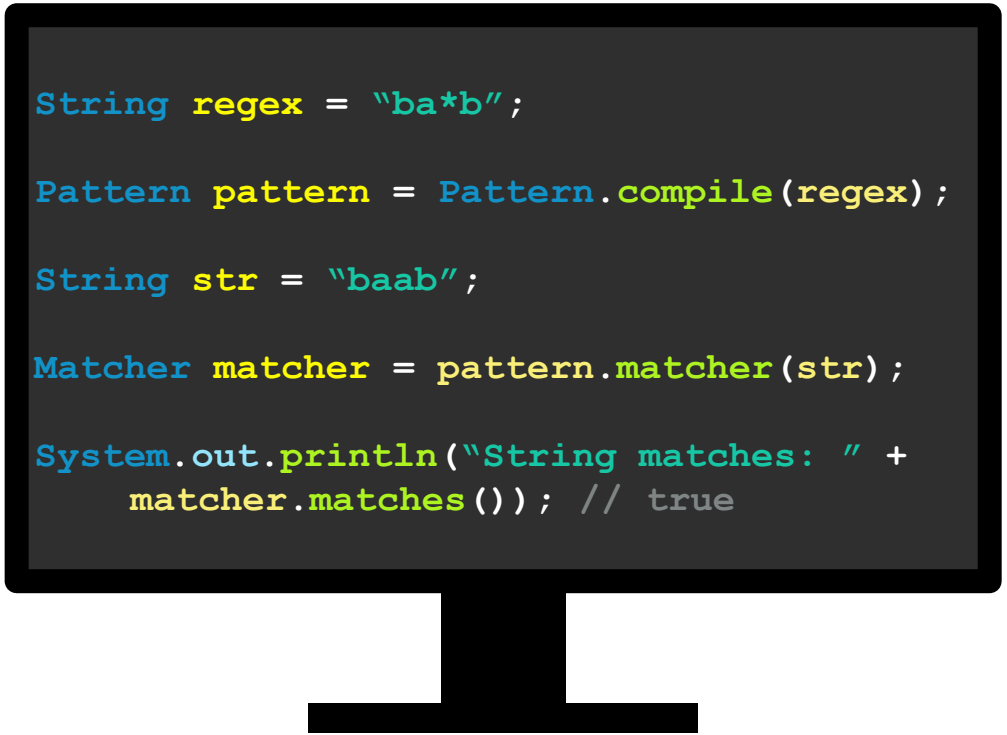
For the **OPTIONAL** exercise, create regex patterns using previous regex examples. Use a regex checker like **https://regex101.com/** to test your answers.

# Optional Regex Exercise

1. Create a regex that matches an email formatted like: [some name]@cognixia.com. The name should only include letters. Examples: Hello@cognixia.com, somename@cognixia.com, etc.
2. Create a regex that matches an area code of 5 digits. Examples: 12345, 20341, etc.
3. Match for a special code that starts with a single capital letter, followed by 4 digits, a single dash, then followed 5 capital letters. Examples: X1234-ABCDE, B2468-XYZAC, etc.
4. Match for a string that cannot contain any vowels. Examples: 123bcd, xyz, BCD!, etc.
5. Match for a string that contains the word “Hello” or a set of 5 capital letters.

# Pattern & Matcher

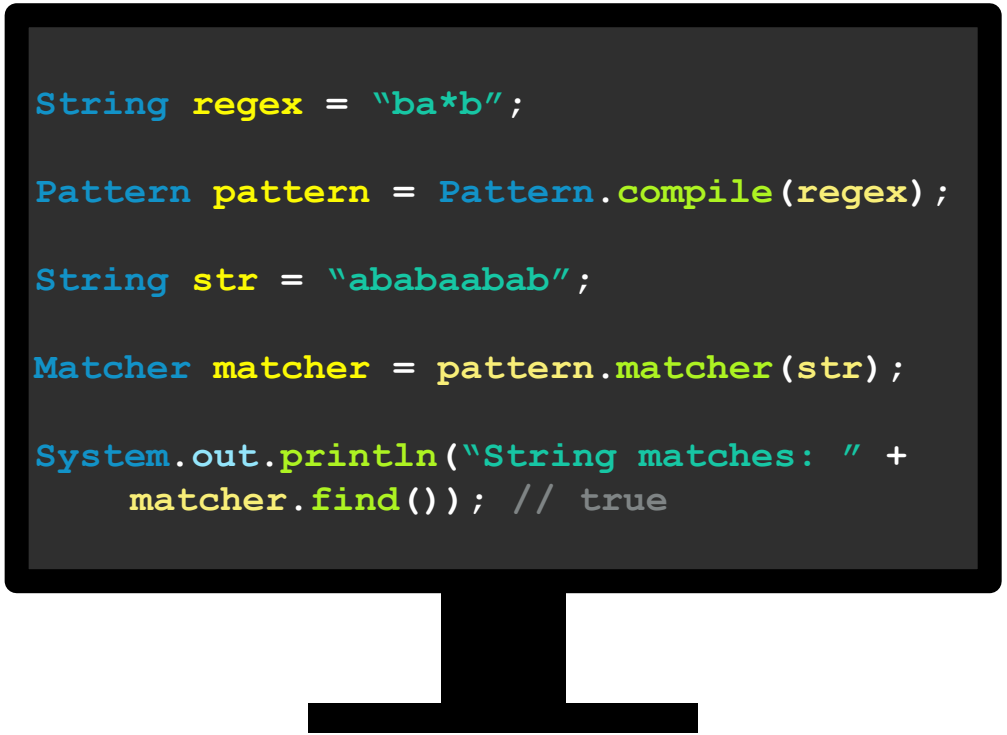
- ➔ **Pattern** = compile representation of a regular expression
- ➔ **Matcher** = created from a Pattern, matches a regular expression against a string



```
String regex = "ba*b";

Pattern pattern = Pattern.compile(regex);

String str = "baab";

Matcher matcher = pattern.matcher(str);

System.out.println("String matches: " +
    matcher.matches()); // true
```

# Pattern & Matcher

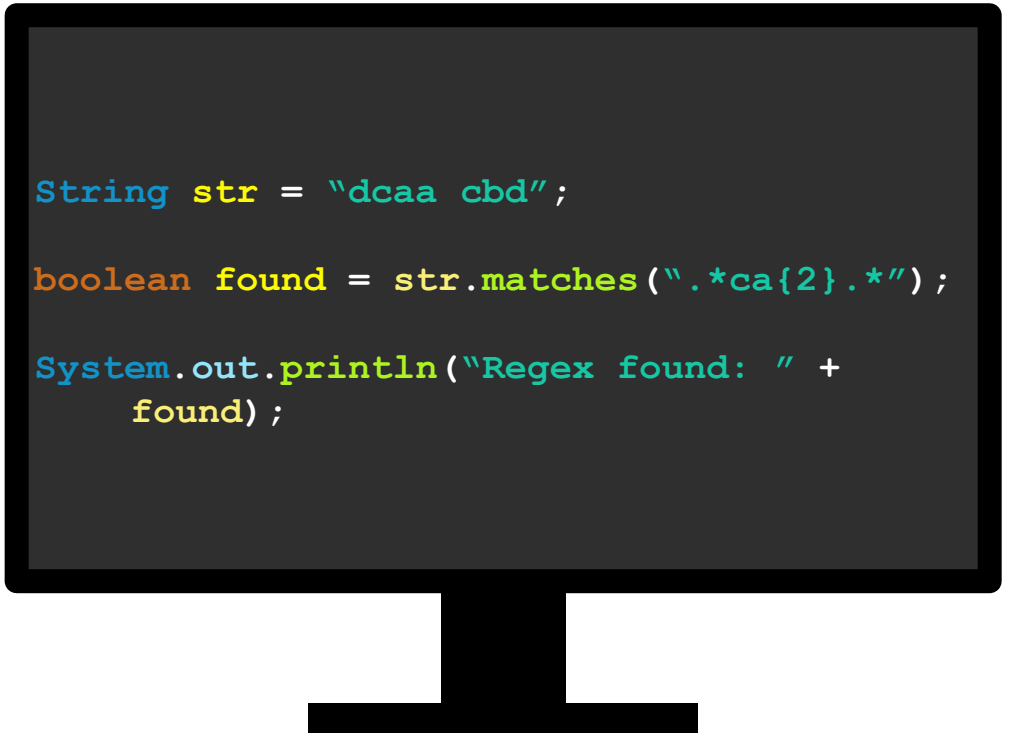
- Use ***matches()*** to match the entire string to the regex (as if there is a ^ at the start and $ at the end)
- ***find()*** is used to match part of a string/substring



```
String regex = "ba*b";

Pattern pattern = Pattern.compile(regex);

String str = "abababaabab";

Matcher matcher = pattern.matcher(str);

System.out.println("String matches: " +
    matcher.find()); // true
```

# Matches Method with String

- **compile** method of ***Pattern*** converts regex to internal format that allows for pattern-matching operations
- **matches** method from ***String*** carries out conversion every time
- ***Pattern should be used if searching for the same regex multiple times to avoid recompiling again and again***



# String: replaceAll()

- ➔ **.replaceAll()** is a String method
- ➔ Accepts a regex and another string, replaces whatever fits that regex with the string given and returns a new modified string

```
String str1 = "abcdefghijklmnopqrstuvwxyz";
String str2 = str1.replaceAll("[aeiou]", "#");
// str2 = #bcd#fgh#jklmn#pqrst#vwxyz
```

# Example Java Code

- Refer to the following Java files to review how to use regex within Java from the previous slides:
  - `PatternAndMatcher.java`
  - `FindMethod.java`
  - `StringRegex.java`
- Optional: Test your patterns from the previous exercise using Java

# Enums

# Enums

- **Enumerated Constants** or **Enums** are a Java language supported way to create constant values
- More type safe than variables like String or int
- If used properly, enums help create reliable and robust programs
- Elements should be named in all uppercase with words separated by underscores ("_")

```
public enum Days {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY
}

public enum Months {
    JANUARY, FEBRUARY, MARCH, APRIL, MAY,
    JUNE, JULY, AUGUST, SEPTEMBER,
    OCTOBER, NOVEMBER, DECEMBER
}
```

# Enums

- Enums create public static final int variables.
- Without enums:

```
public static final int ICE_CREAM = 0;
public static final int CHOCOLATE = 1;
public static final int VANILLA = 2;
```

- With enums:

```
public static enum Cake {
    ICE_CREAM, CHOCOLATE, VANILLA
}
```

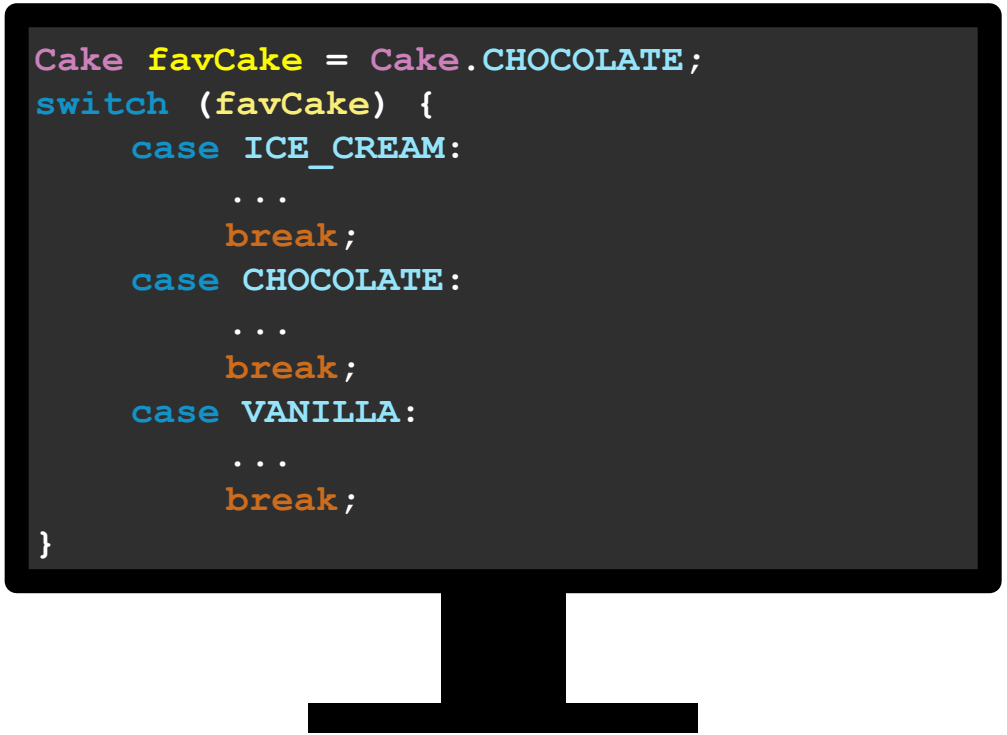
# Enums

- ➔ Once an enum is created, its values can be accessed with dot notation
- ➔ A local variable can be instantiated of the enum's type, then initialized with a value from the enum :

```
public static enum Cake {
    ICE_CREAM, CHOCOLATE, VANILLA
}
...
Cake c1 = Cake.CHOCOLATE;
System.out.println(c1); // CHOCOLATE
```

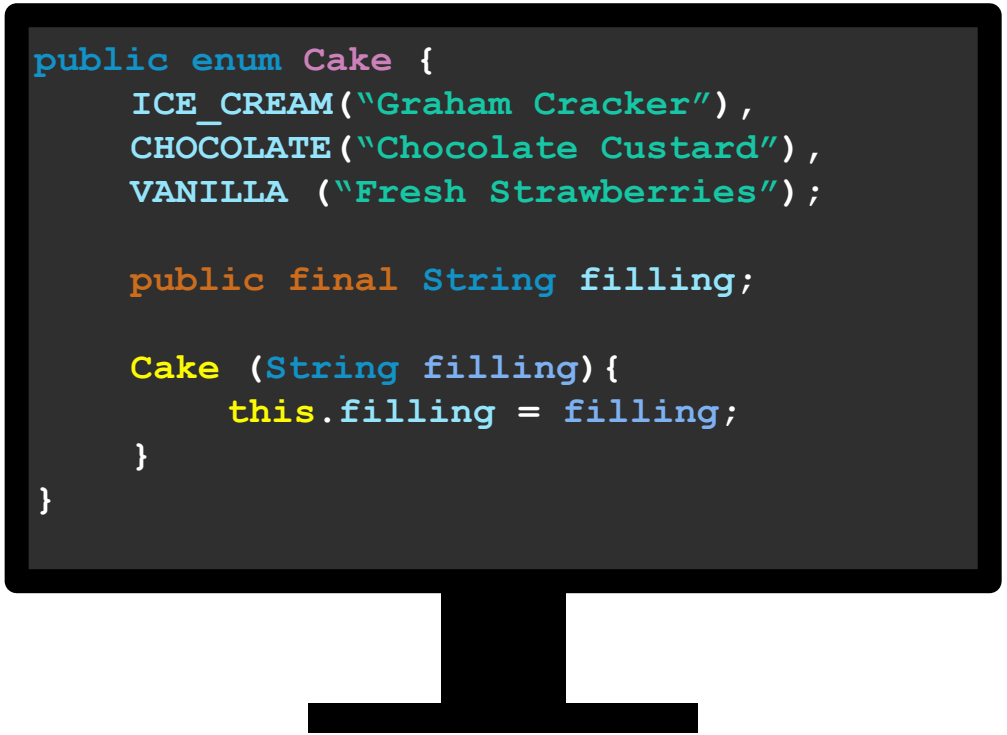
# Enums in Switches

- ➔ Enums can be used in switch cases.
- ➔ Enums allow for switch cases to use values no more complex than int or char



# Enums with Constructors

- Each value in an enum instantiates a public final class
- The constructor defined in the enum is run for each value
- Arguments passed to the enumerated values can be used in the constructor to create new data members



# Where Should I Be Using an Enum?

- Please review the following video on enums: https://youtu.be/8Bz9VQxK6vg
- Feel free to review attached code used in video:
  - `EnumsExample.java`
  - `StudentString.java`
  - `StudentInt.java`
  - `StudentEnum.java`
- Following slides in this section can be used as additional reference

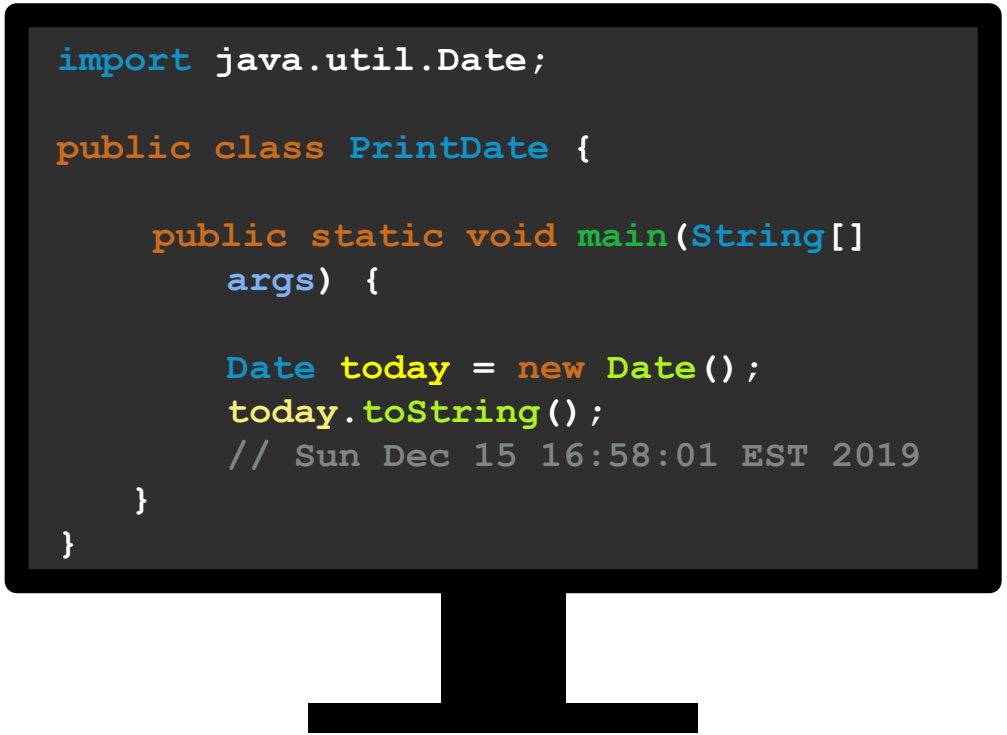
# Example Java Code

- Besides the files in the video, review the following code on enums to further understand their properties:
  - `EnumsExample.java`
  - `Element.java`
- Make sure to uncomment some of the methods that contain the examples

# Dates

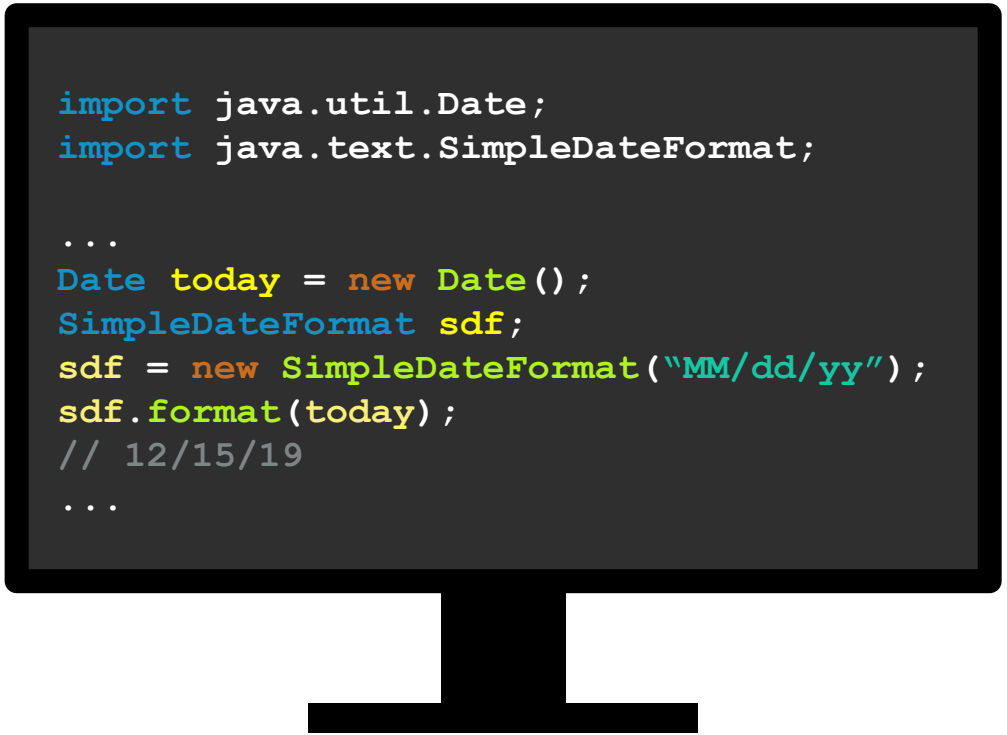
# java.util.Date

- Java comes with **Date** package, imported from **java.util**
- New Date object created is given current date from system
- Though it displays a timezone in its toString(), does not store one internally, grabs this information from system



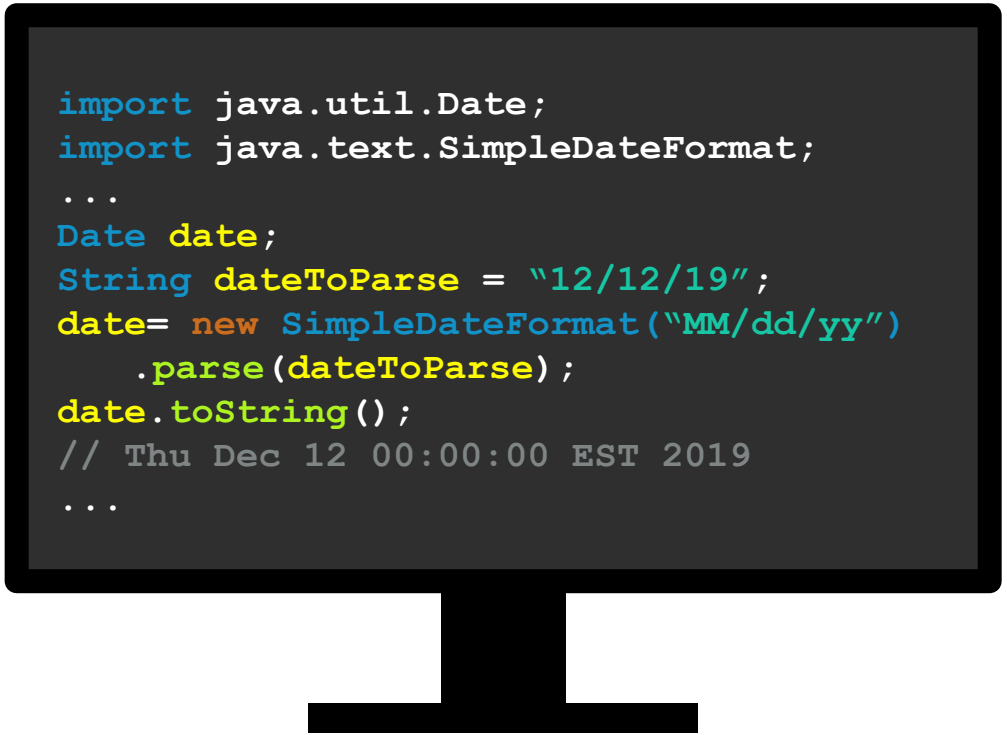
# Formatting Dates

- ➔ **SimpleDateFormat** can be used to change how the date is presented
- ➔ By passing desired format (ex. Month/Day/Year) in the ***SDF*** constructor, ***Date*** object can be displayed with any desired format



# String to Date

- **SimpleDateFormat** can also convert date string to Date object
- The string “12/12/19” is converted to date object
- SDF object is created to parse incoming string to useable date



# Java 8 Introduced New Date Classes

- Java 8 introduced **Date and Time API** to replace the **Date and Calendar API**. The Date/Time API makes improvements in three key areas:
  - **Thread Safety**
    - The Date/Calendar classes were not thread safe. The Date/Time classes are thread safe and immutable
  - **Ease of Understanding**
    - The Date/Time API offers more methods for common use cases
  - **Time Zones**
    - The Date/Time API added the *Zoned* and *Local* classes to handle time zones automatically

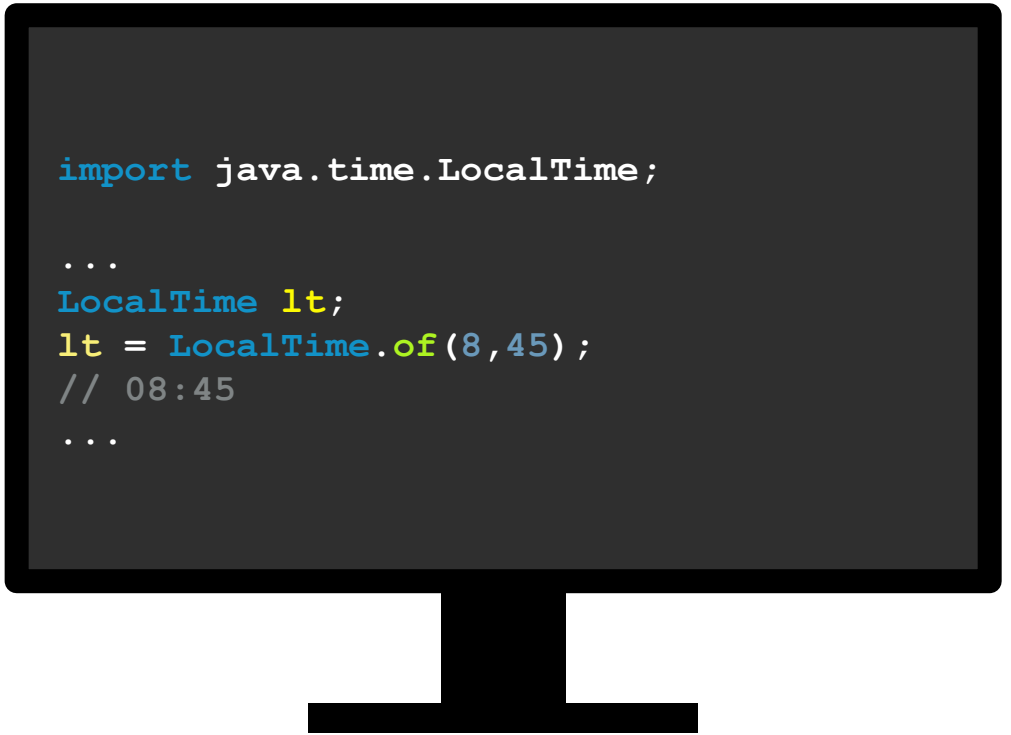
# LocalDate

- **LocalDate** used to represent a date when time zones do not need to be considered
- By passing in ***year, month, and day;*** a new LocalDate object is created
- LocalDate gives access to useful methods associated with dates
  - Adding days, months, or years
  - Checking the day of the week
  - Checking if one date is before another

```
import java.time.LocalDate;

...
LocalDate ld;
ld = LocalDate.of(2015,7,3);
// 2015-07-03
...
```

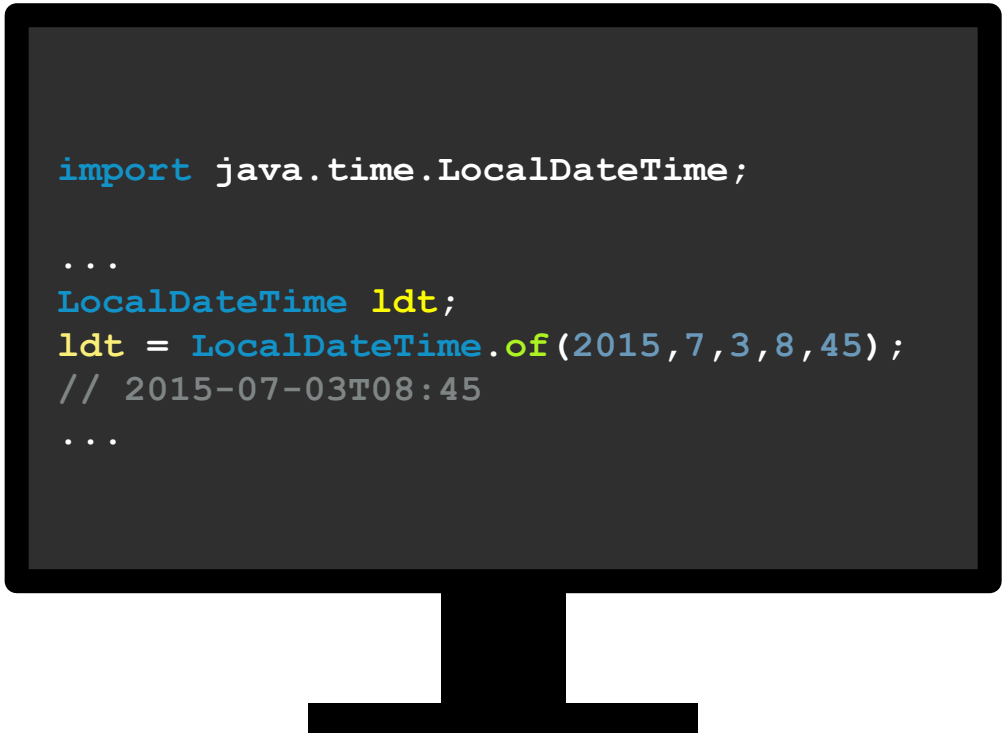
# LocalTime

- **LocalTime** used to represent a time without a date
- By passing in an ***hour and minute***; a new LocalTime object is created
- LocalTime gives access to useful methods associated with times
  - Adding hours or minutes
  - Checking if a time is before another



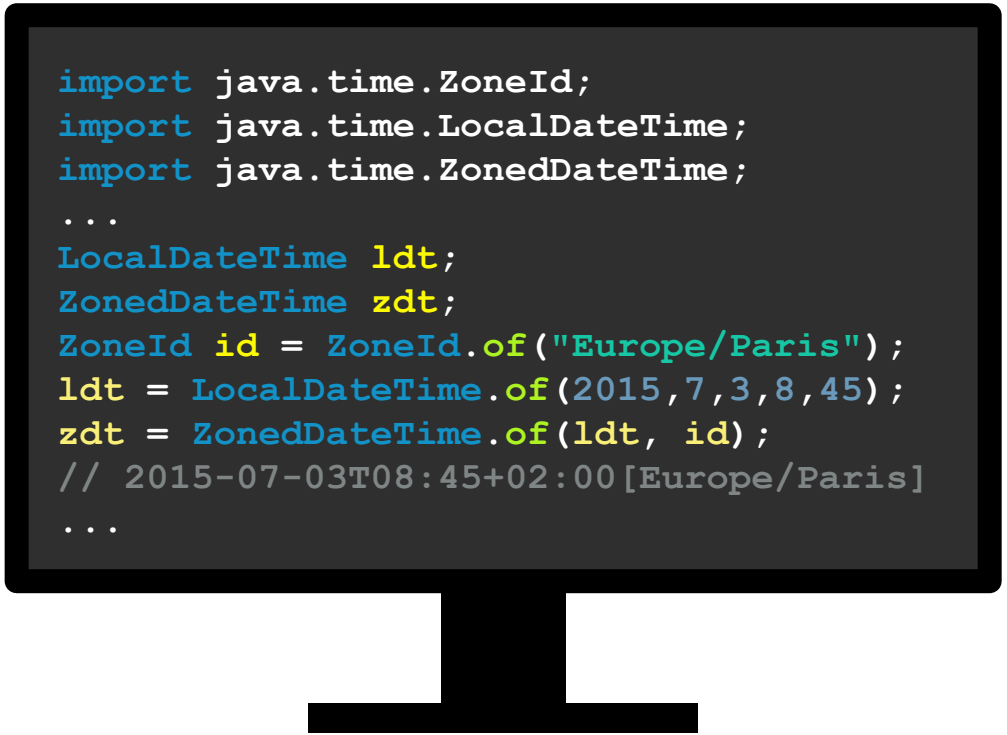
# LocalDateTime

- **LocalDateTime** used to represent a combination of date and time
- By passing in ***year, month, day,hour, and minute***; a new LocalDateTime object is created



# ZonedDateTime

- **ZonedDateTime** used to create an object representing a date in a given time zone
- **ZoneId** object created with the desired time zone
- ZonedDateTime created with the local date and time and the ZoneId.

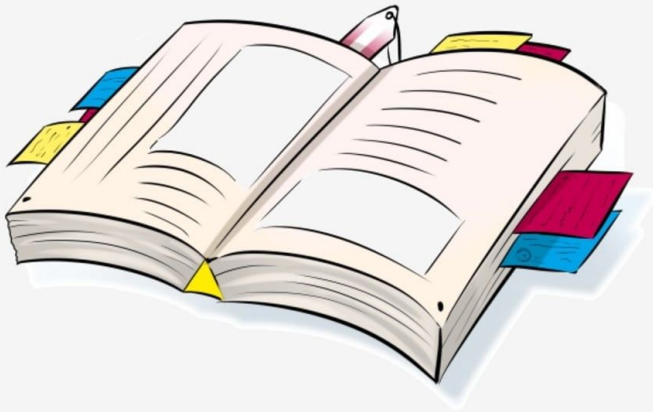


# Dates Exercises

1. Convert the string "March 12, 2012" to a Date object
2. Create a small program that **accepts a date from a user** (either a string or asking for day, month, year one by one). Then prompt the user **how many days they wish to add to this date**
    a. Example: User gives date of ***July 3, 2019*** and adds 10 days so the date printed is ***July 13, 2019***

# Open Book Quiz on Regex, Enums, and Dates

- ➔ https://forms.gle/yHasSHbEvjdKuXUF9
- ➔ This is an **open note**, multiple choice quiz
- ➔ Have it completed by the **start of class tomorrow at 10AM EST**
- ➔ If there are ***any questions, ask your instructor during this time or during office hours***, as they may not able available after hours




Cognixia™