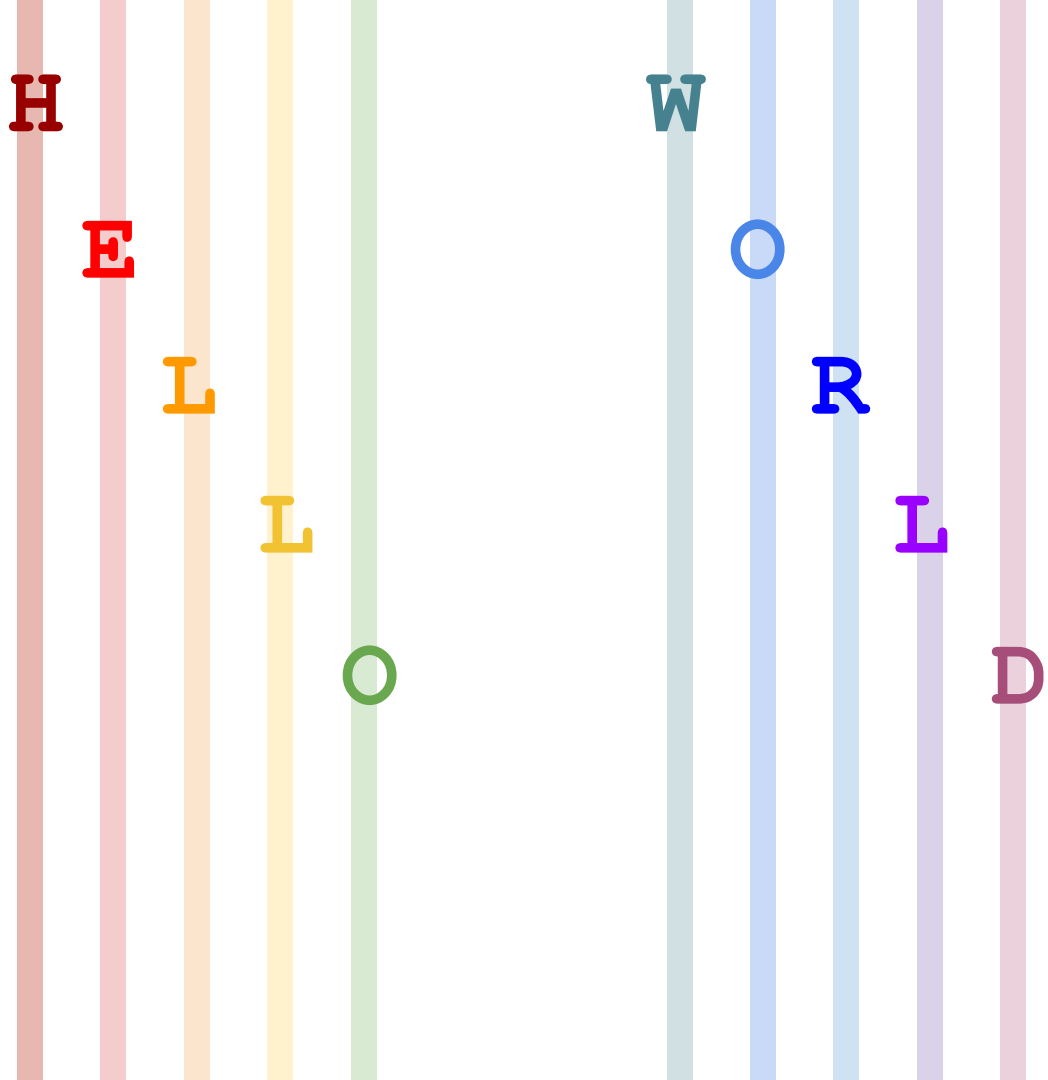
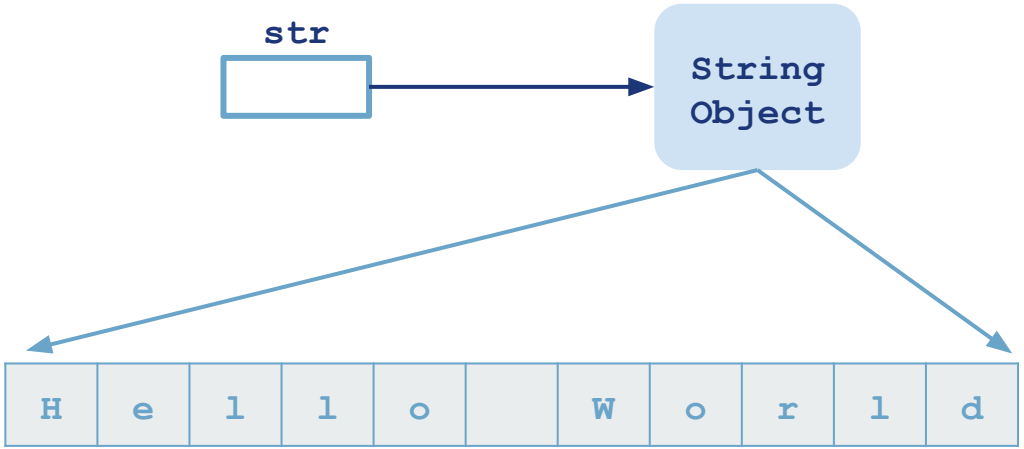
Strings



Strings

- ➔ **Strings** are not primitive data types, they are objects
- ➔ New object created with name assigned to its memory location
- ➔ Often considered alongside primitives even though they are objects

```
String str =  "Hello World";
```



String Concatenation

- The **+ operator** can concat a string with other data types
  - Will convert other data types to strings
- Using **concat()** method
  - Doesn't edit string, returns new string with concatenation
  - Only accepts Strings
  - Will throw exception if String passed is null

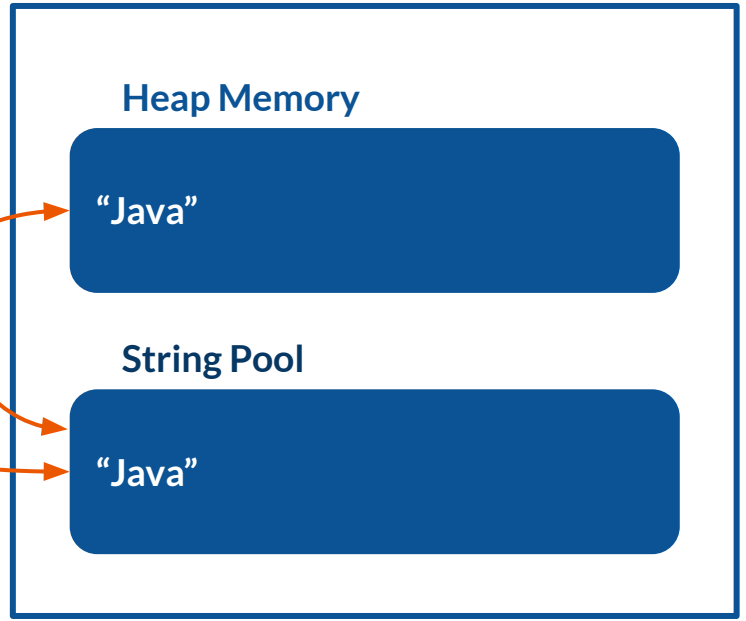
```
String str1 =  "Hello ";

// str2 = "Hello World"
String str2 = str1.concat("World");

boolean bool = true;

// str3 = "Hello There2.3true"
String str3 = str1 + "There" + 2.3 +
bool;
```

String Immutability and String Pool

- ➔ Strings are ***final and immutable*** — can't be altered (adding/removing a character)
- ➔ New object created each time value reassigned
- ➔ **String Pool** — place in memory where the JVM stores each unique character sequence that is created
- ➔ When a new string is created,
  - ◆ JVM searches for that char sequence in String Pool
  - ◆ Sequence found, assigns new object that value in the pool
  - ◆ Not found, creates char sequence in String Pool and assigns the variable to that memory location
- ➔ Using new keyword → String stored in heap memory

String Pool Diagram

```
String str1 =  "Java";

String str2 = new String("Java");

String str3 =  "Java";
```



Comparing Strings

- Consider the following scenario:

```
String str1 =  "Java";
String str2 = new String( "Java");
```

- Two strings have the same char sequence. However, if we try and compare them:

```
str1 == str1; // true
str2 == str1; // false
```

- Strings are objects, so the variables we assign them are references to memory locations, not the char sequences themselves.
- To check the equivalence of strings, we must use the .equals( ) method:

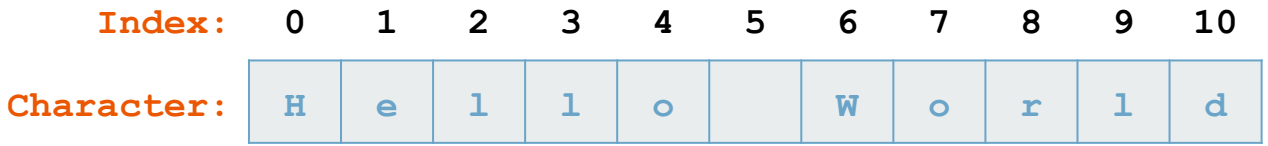
```
str1.equals(str1); // true
str2.equals(str1); // true
```

String Methods

There are almost 50 methods defined in the String class. Important methods include:

- **Length**:
  - Returns the number of chars in the string as an int
- **CharAt**:
  - Returns the char at the given index
- **IndexOf**:
  - Returns the first index of a given char in the string
- **Substring**:
  - Extracts a section of the string at the given indexes
- String **Concatenation**, while not a defined method, is an important operation when working with strings

Length of a String

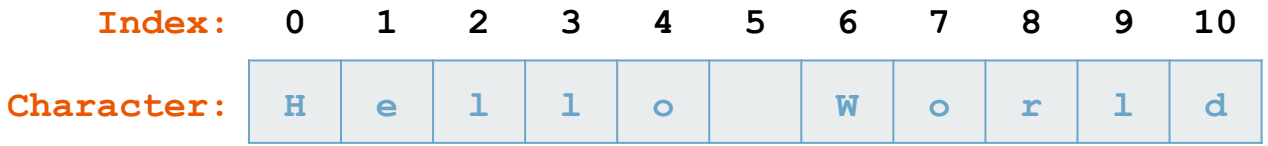


- Strings are arrays of characters (more on arrays later)
- ***length( )*** returns number of characters in the array

```
System.out.println("string length:  " + str.length());
// string length: 11
```

- Important: Array indexes start at 0, so the length of the string is one more than the last index

Single Character Access

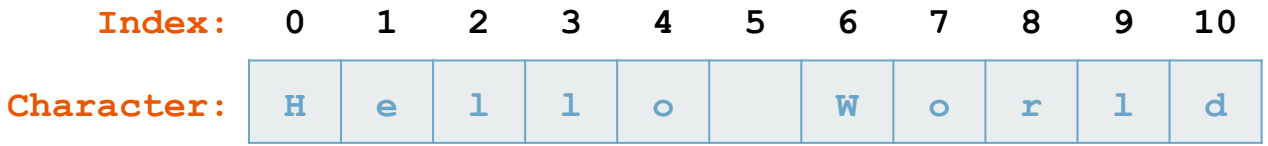


➔ Individual characters can be accessed with the ***charAt()***:

```
System.out.println("first letter:  " + str.charAt(0));
// first letter: H
```

➔ Trying to access a char outside the length of an array results in an error

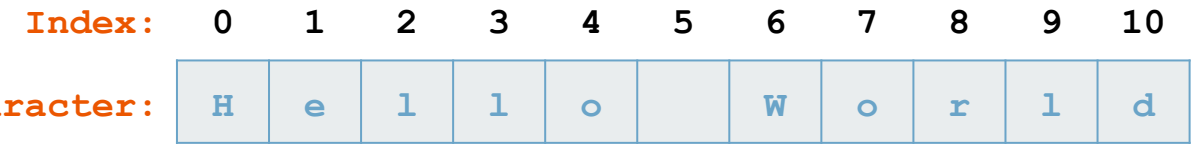
Finding an Index



- ***indexOf( )*** method returns index where a given substring appears in the string
- Is overloaded operator
  - can accept a string or a char
  - takes second argument of a starting index

```
String s1 = "first letter 'H': " + str.indexOf('H'); // first letter 'H': 0
String s2 = "first 'W': " + str.indexOf('W'); // first letter 'W': 6
String s3 = "'o' after index 5: " + str.indexOf('o', 5); // 'o' after index 5: 7
```

Substrings



- ***substring( )*** retrieves specific part of a string
- Returns new string, original string not modified
- Overloaded, accepts one or two arguments
  - *.substring(i, j)* return new string starting with character at ***i*** and ending before index ***j***
  - *.substring(i)* returns new string starting at index ***i*** till the end of string

```
String s1 = str.substring(3, 7); // "lo W"
String s2 = str.substring(4); // "o World"
```

Other String Methods

- **compareTo( )**
  - Compares two strings lexicographically
- **trim( )**
  - Removes leading and trailing whitespace characters
- **valueOf( )**
  - Converts a primitive to a string
- **startsWith( )**
  - Returns true if the string starts with the given char
- **endsWith( )**
  - Returns true if the string ends with the given char
- Most IDEs will give a user access to all available methods for a given class.
  - String methods detailed in Java Documentation: https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/String.html

StringBuffer and StringBuilder

- ➔ **StringBuffer** or the newer **StringBuilder** (Java 5) used to work with mutable character sequences
  - StringBuffer is thread safe, but Stringbuilder is faster.
  - Both have the same methods, interchangeable
- ➔ Contain methods for:
  - Replacing chars
  - Appending or prepending any primitive data type
  - Inserting new char sequences at a given position

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World"); // Hello World
sb.setCharAt(5,'@'); // Hello@World
sb.append(1);        // Hello@World1
sb.append(true);     // Hello@World1true
```


WHITE
BOARD
EXERCISE

Coding challenge:

Create a method that takes a string, reverses it, and returns the reversed string. You can use a built in method to do this

Part 1:

Create a second method that does not use any built in methods to reverse the string.

Part 2:

For a given string with multiple words, reverse each word individually

Regular Expressions

The universal language for character pattern matching

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.

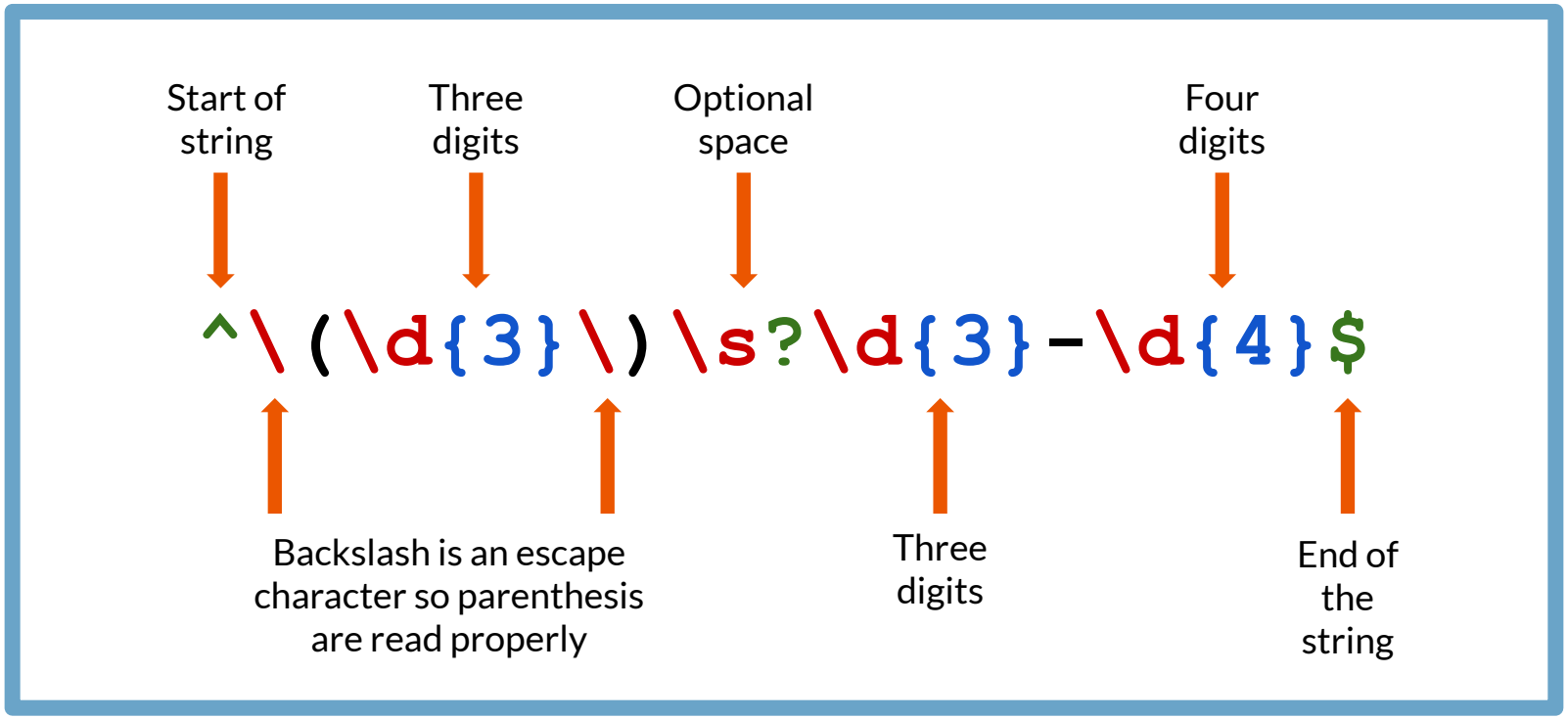
```
^\(\d{3}\)\s?\d{3}-\d{4}$
```

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

^	If first in the regex, denotes the start or as a negation	`^Hello` → string must start with word “Hello” `[^a]` → string cannot have the character “a”
$	Denotes the end of a string	`Hello$` → string must end with word “Hello”
*	Zero or more occurences	`ba*b` → string has two b’s with zero or more a’s between them
+	One or more occurences	`b(ac)+b` → string has two b’s with one “ac” r multiple “ac” strings between them
?	Zero or one occurrence	`bc?b` → string has two b’s with nothing between them or a single c
{}	Matches a string in the range specified	`abc{2, 5}` → string starts with ab and follows with two to five c’s
() and |	Used to provide section of choices	`b(a|i|e)t` → string is either bat, bit, or bet
[]	Same as above	`b[aie]t` → string is either bat, bit, or bet

.	Any character, can be a alphanumeric or a symbol	`b.c` → string can bac, b5c, b@c, etc.
[0-9]	Digit from 0 t 9	`[0-9][0-9]` → string from 00 to 99
[a-z]	Lowercase letter, switch to capital A and Z for all uppercase letters	`[a-z]+` → string with a lowercase characters repeated one or more times
[a-zA-Z]	A letter, lowercase or uppercase	`[a-zA-Z]{2}` → a two character string with any letter (wE, BP, ee...)
\w	A word character: letter, number, or an underscore	`\w@gmail.com` → hello@gmail.com, 123@gmail.com, b_tr@gmai.com
\d	A digit	`\d*` → string with zero or more digits (4, 48, 489..)
\s	Whitespace character that includes tabs and line breaks	`hello\sworld` → string has a whitespace character like a tab or line break between the two words

Additional resources here and here.


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

Reg(Ex) Exercise

Please close your laptops, have a paper and writing utensil out. Make sure you have received the regex reference sheet. When you have your answer, raise your hand.

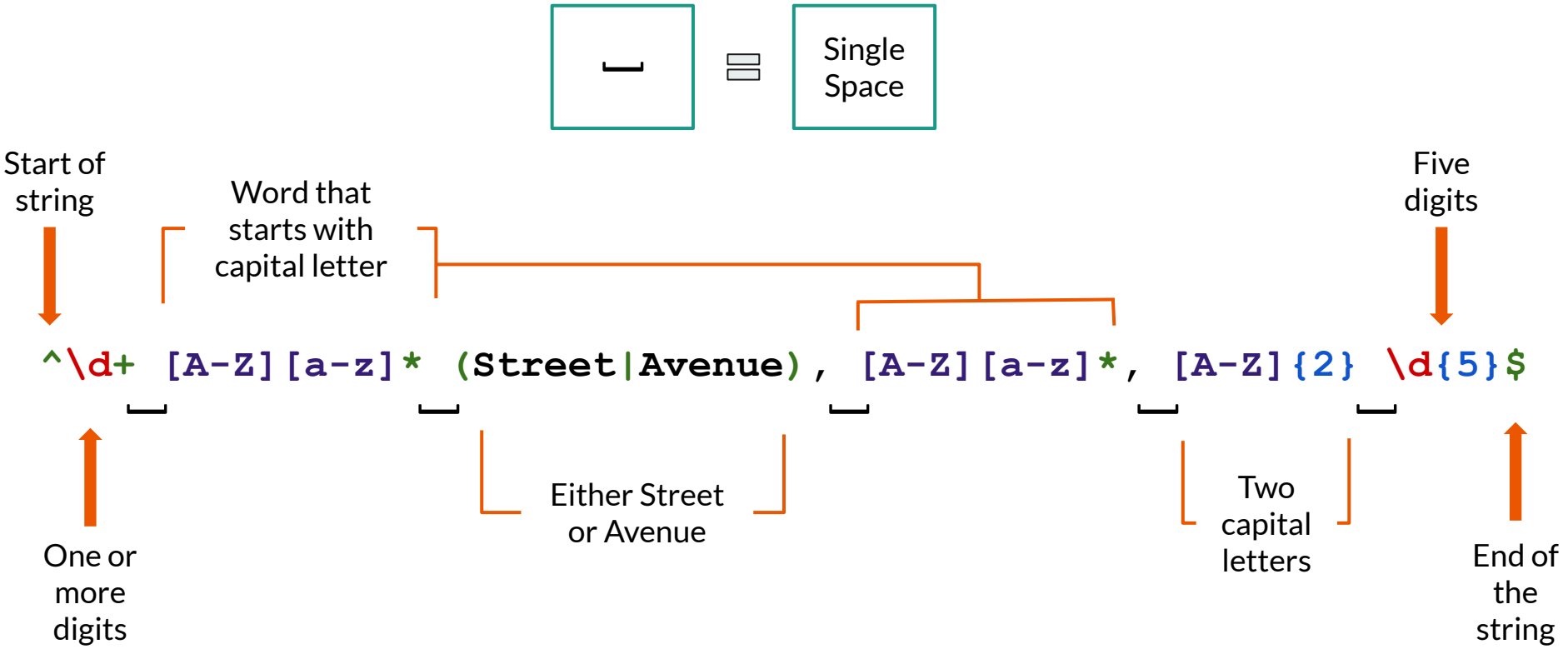
Write a regex expression for an address. For this exercise we will assume that an address is formatted like this:

1 Some Street, City, ST 12345

The street just has to lead with a number. Street name should be two words with only letters. The second word of the street name should be either “Street” or “Avenue”. The city should be one word. The state should be two capital letters, doesn't have to be a real state. The zip can just be exactly 5 digits long. Have a comma before and after the city.

25 Water Avenue, Bee, BZ 45092 ✓

9 South 27th Place, A City, CAB 12378-0123 ✗


␣
Single Space
Start of string
Word that starts with capital letter
Five digits
^\d+ [A-Z][a-z]* (Street|Avenue), [A-Z][a-z]*, [A-Z]{2} \d{5}$
One or more digits
Either Street or Avenue
Two capital letters
End of the string

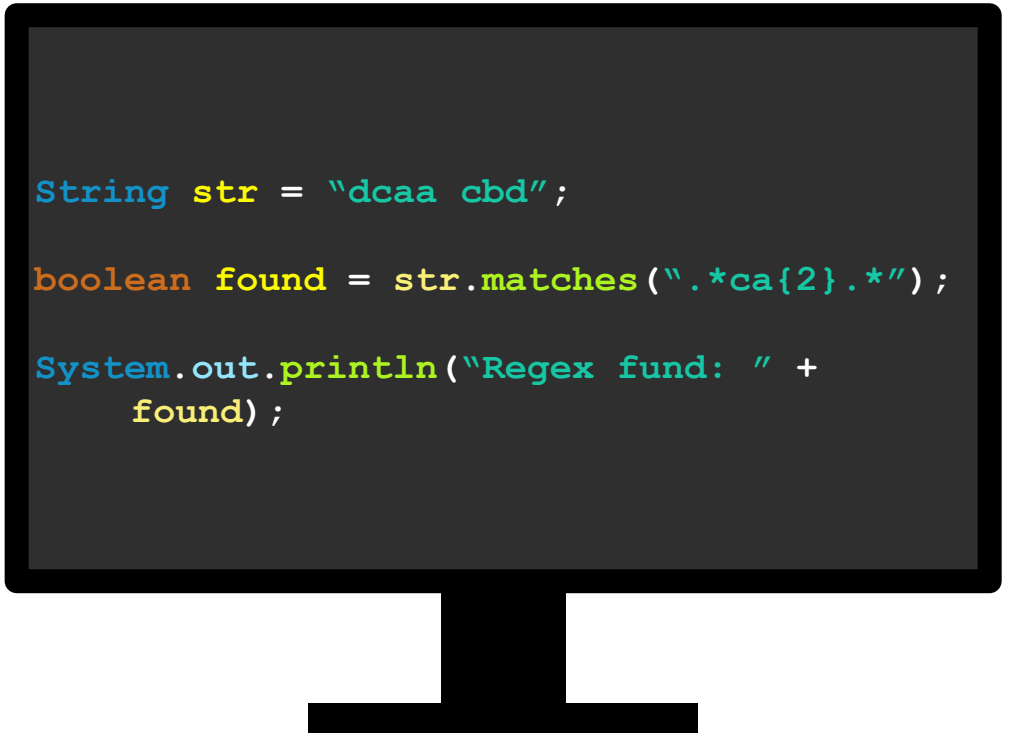
Pattern & Matcher

- ➔ **Pattern** = compile representation of a regular expression
- ➔ **Matcher** = created from a Pattern, matches a regular expression against a string
- ➔ Use ***matches()*** vs ***find()*** for Matcher depending on if you want to match regex to entire string or part of it

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

Pattern pattern = Pattern.compile(regex);

String str = "baab";

Matcher matcher = pattern.matcher(str);

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

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
- ➔ Use Pattern if searching for same pattern multiple times

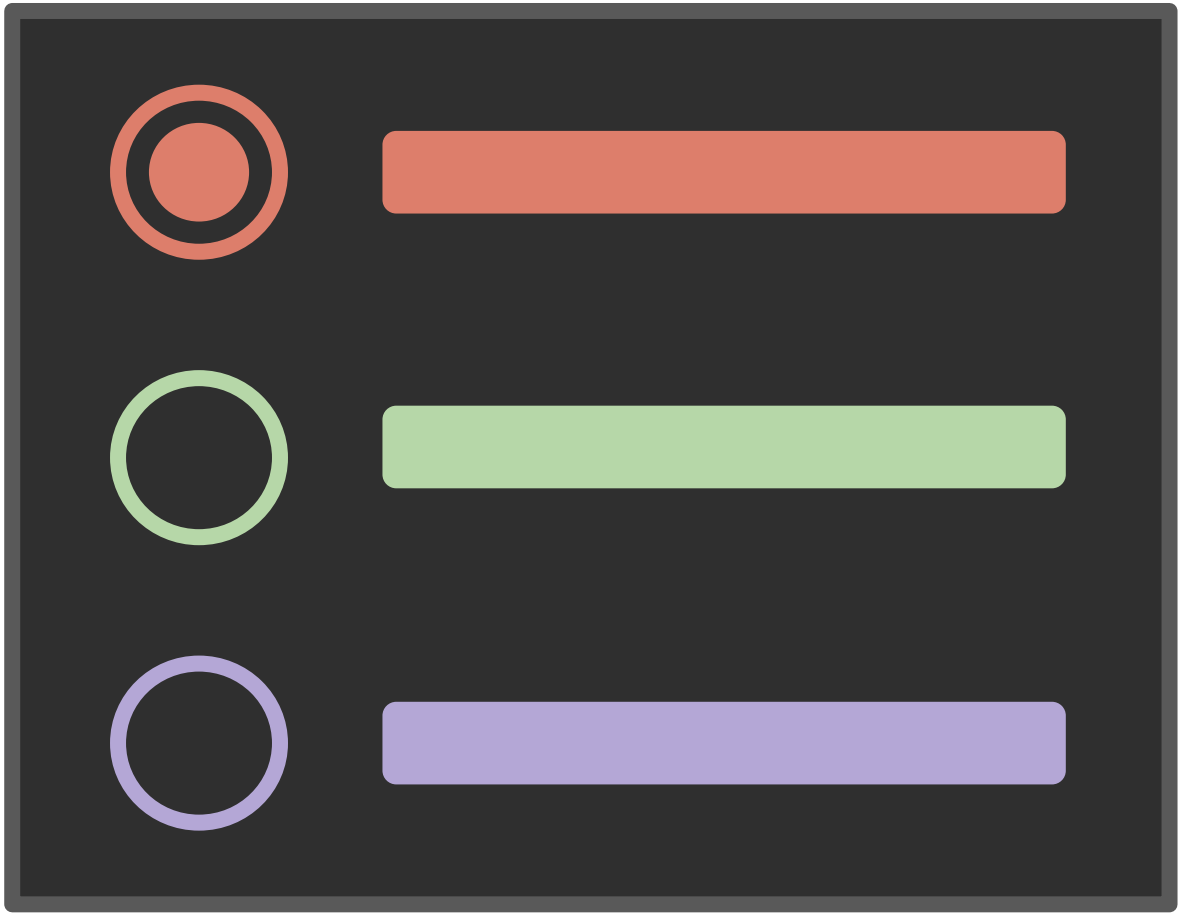


String: replaceAll()

- ➔ **.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
```

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
}
```

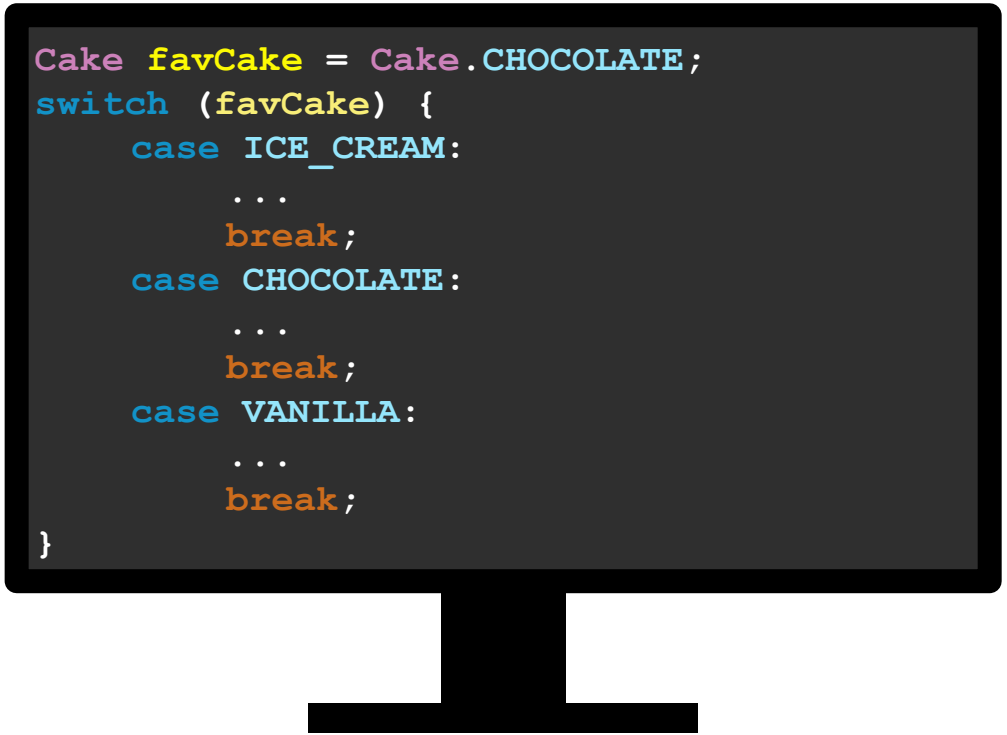
Enums

- ➔ 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
```

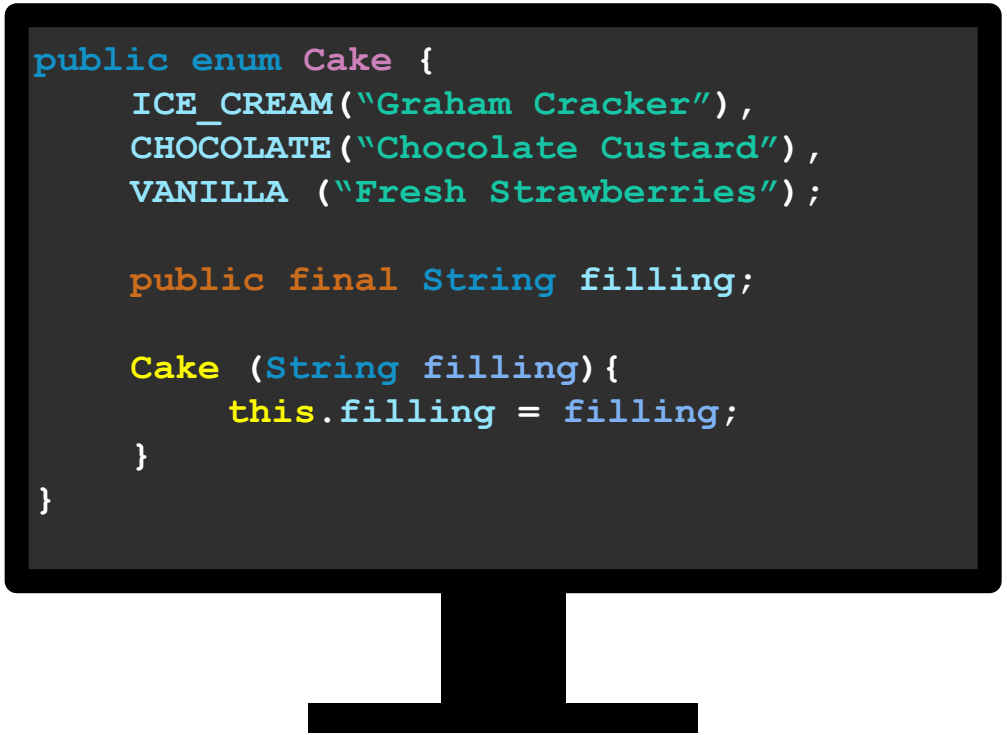
Enums in Switches

- ➔ 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

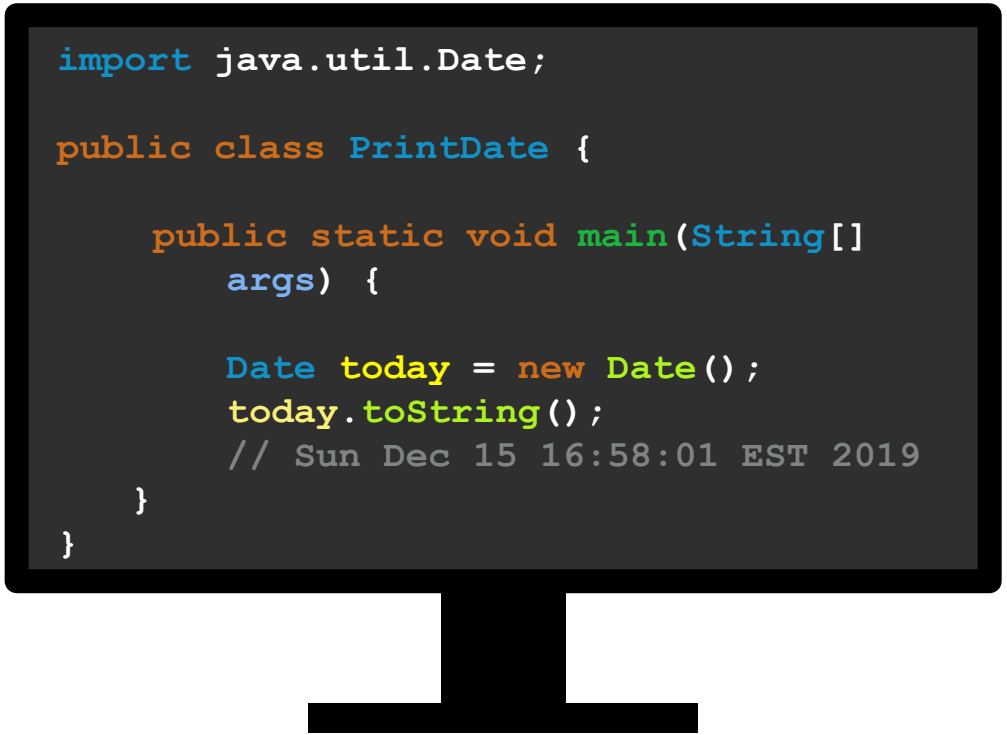


Dates


SEP
Su Mo Tu We Th Fr Sa
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

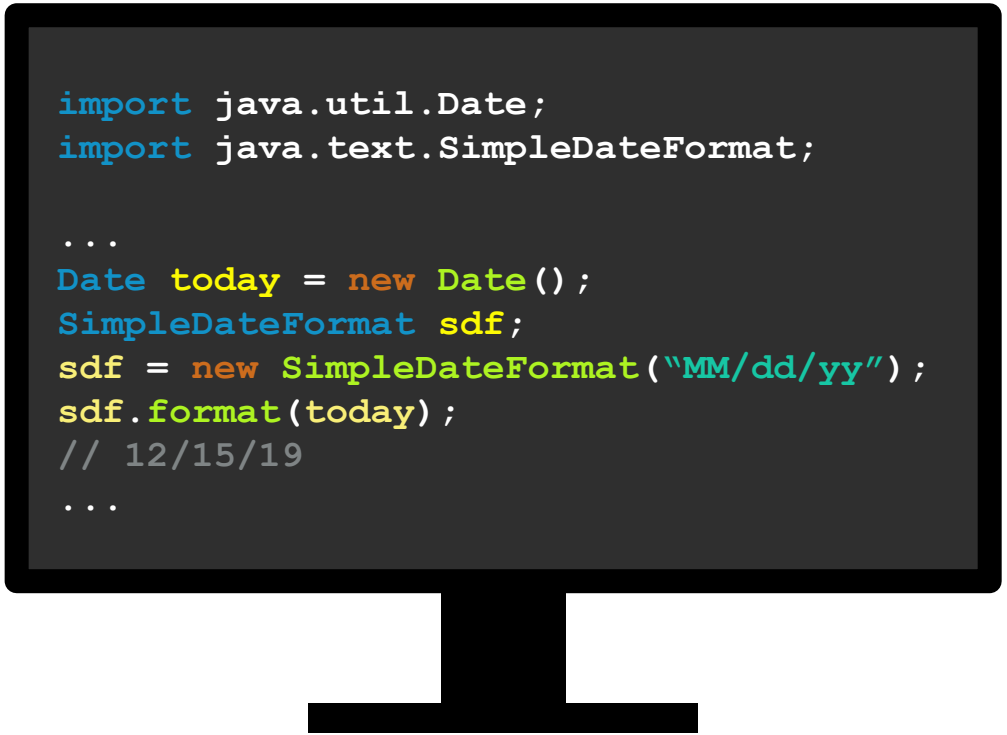
java.util.Date

- Java comes with **Date** package, imported from **java.util**
- New Date object is created, given current date from system
- Date objects contain a toString( ) method, converts date to readable string



Formatting Dates

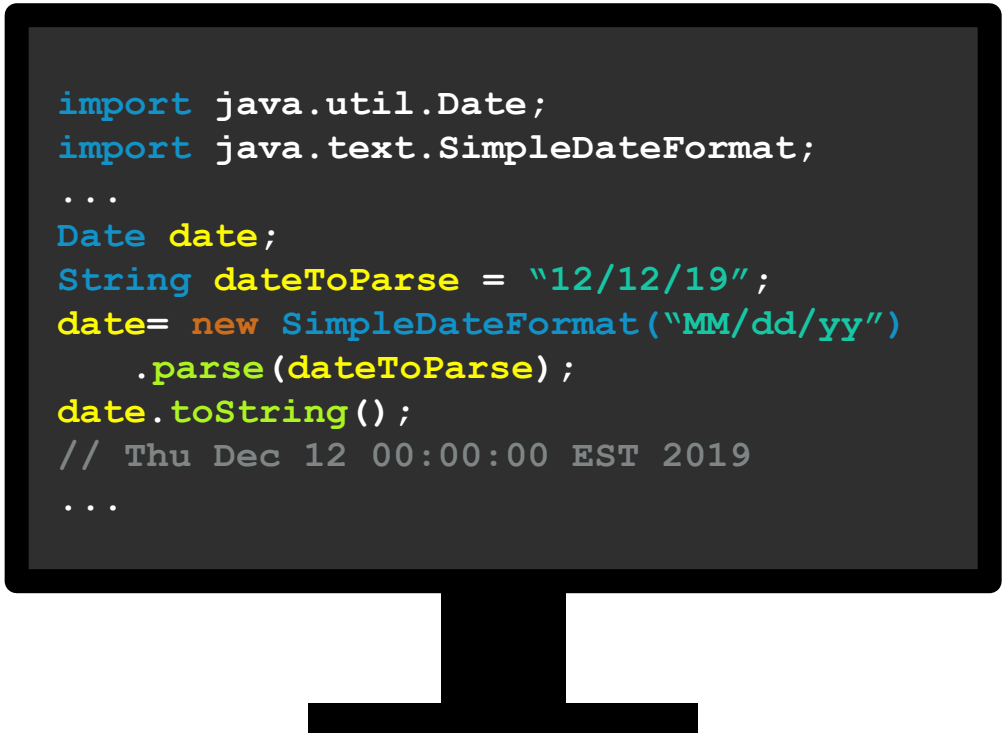
- ➔ **SimpleDateFormat** can be used to change how the date is presented
- ➔ By passing desired format (ex. Month/Day/Year) in the ***SDF*** constructor, ***Date*** object can be displayed with any desired format



```
import java.util.Date;
import java.text.SimpleDateFormat;

...
Date today = new Date();
SimpleDateFormat sdf;
sdf = new SimpleDateFormat("MM/dd/yy");
sdf.format(today);
// 12/15/19
...
```

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

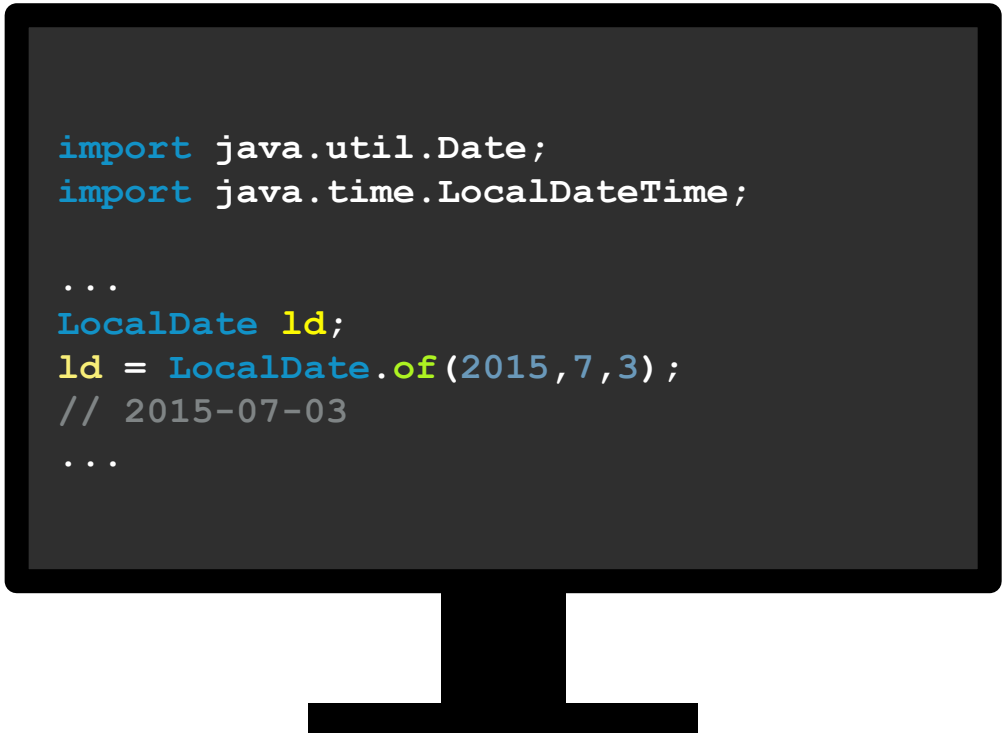


Java 8 Introduced New Date Classes

- 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

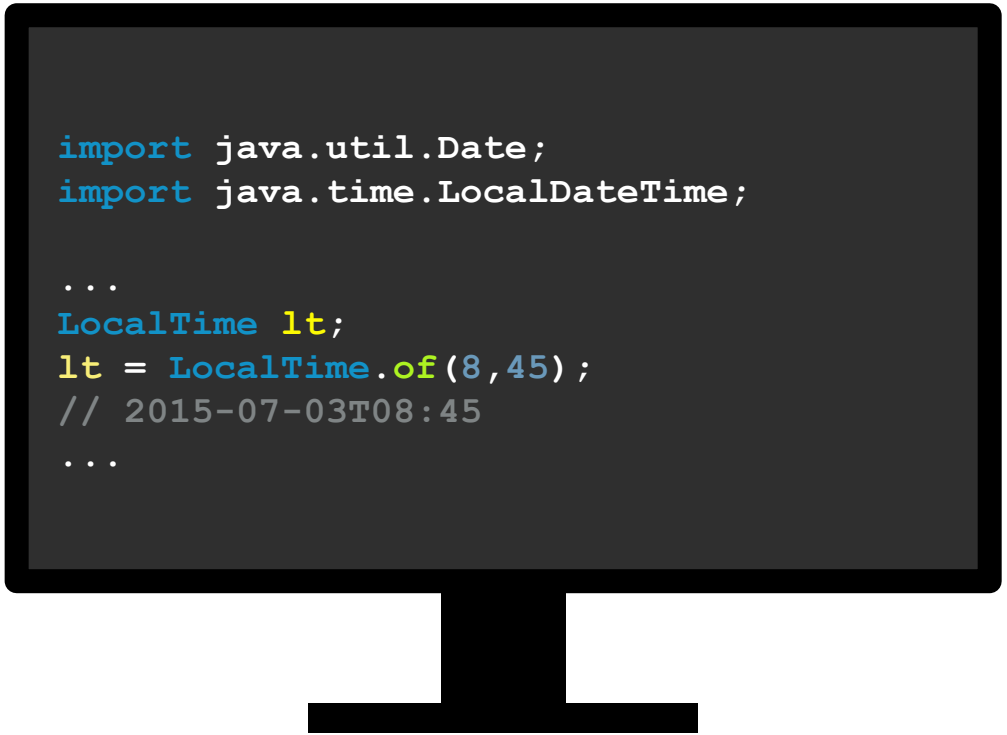
- **LocalDate** used to represent a date when time zones do not need to be considered
- By passing in ***year, month, and day;*** a new LocalDate object is created
- LocalDate gives access to useful methods associated with dates
  - Adding days, months, or years
  - Checking the day of the week
  - Checking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



```
import java.util.Date;
import java.time.LocalDateTime;

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

LocalTime

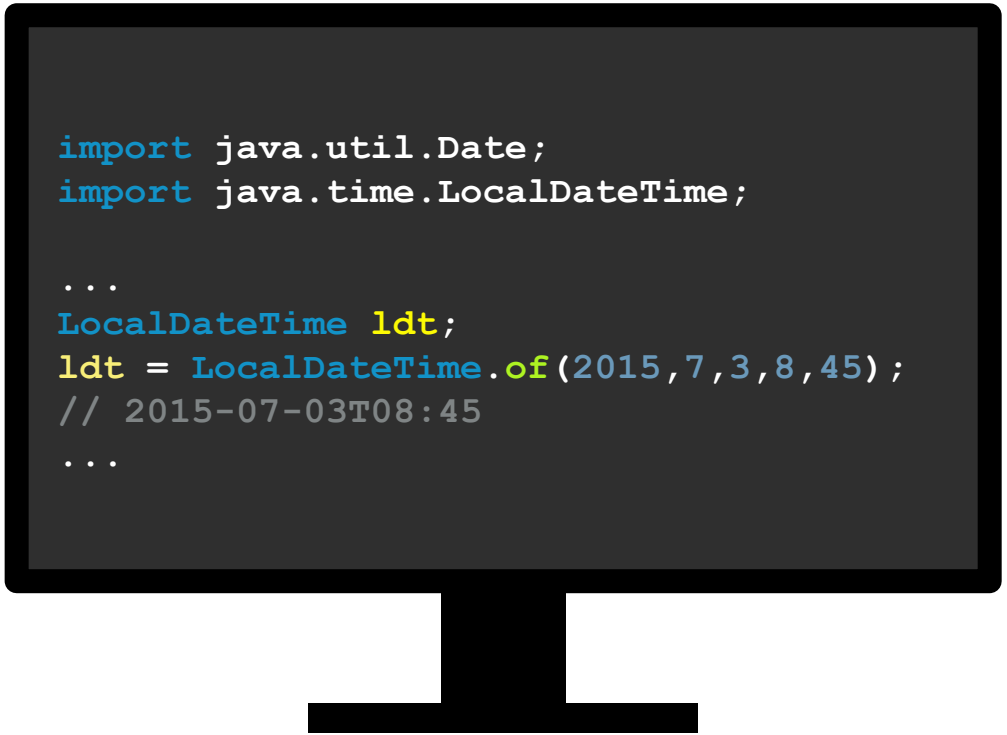
- **LocalTime** used to represent a time without a date
- By passing in an ***hour and minute***; a new LocalTime object is created
- LocalTime gives access to useful methods associated with times
  - Adding hours or minutes
  - Checking if a time is before another



```
import java.util.Date;
import java.time.LocalDateTime;

...
LocalTime lt;
lt = LocalTime.of(8,45);
// 2015-07-03T08:45
...
```

LocalDateTime

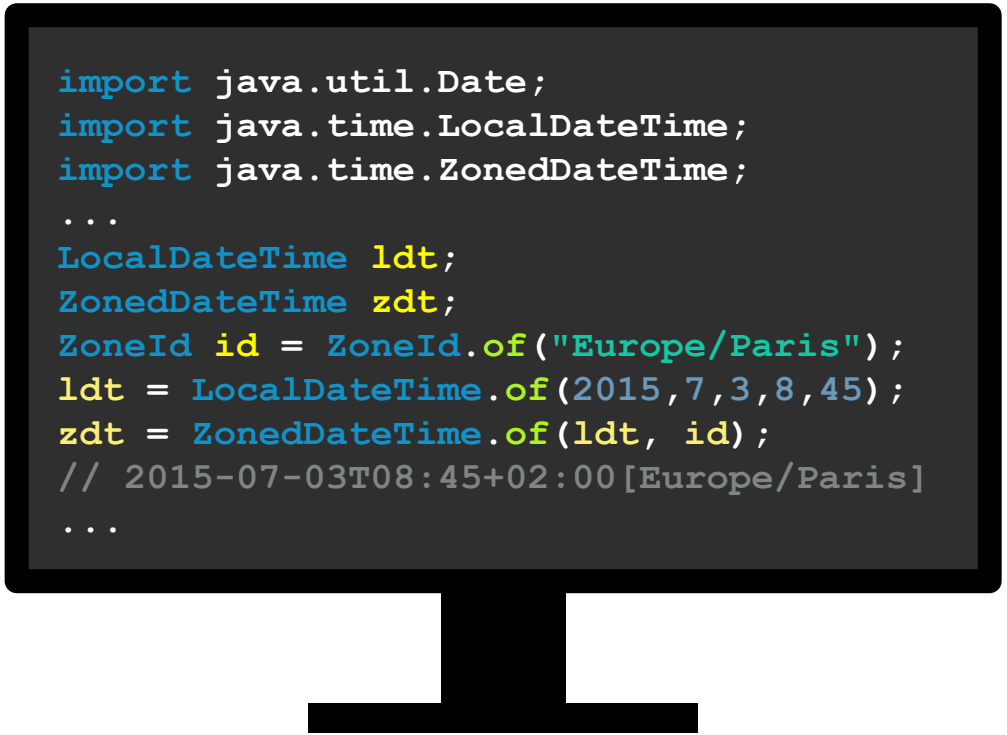
- **LocalDateTime** used to represent a combination of date and time
- By passing in ***year, month, day,hour, and minute***; a new LocalDateTime object is created



```
import java.util.Date;
import java.time.LocalDateTime;

...
LocalDateTime ldt;
ldt = LocalDateTime.of(2015,7,3,8,45);
// 2015-07-03T08:45
...
```

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.



```
import java.util.Date;
import java.time.LocalDateTime;
import java.time.ZonedDateTime;
...
LocalDateTime ldt;
ZonedDateTime zdt;
ZoneId id = ZoneId.of("Europe/Paris");
ldt = LocalDateTime.of(2015,7,3,8,45);
zdt = ZonedDateTime.of(ldt, id);
// 2015-07-03T08:45+02:00[Europe/Paris]
...
```

Convert the following Strings to Date objects:

04/23/2004

November 05 2014

1993/30/09

Assume the following Dates and Times on the left are in the EST time zone. Convert them to the given time zones on the right.

04/23/2004 7:30pm ➡ **Paris**

2013/30/09 21:50 ➡ **Honolulu**

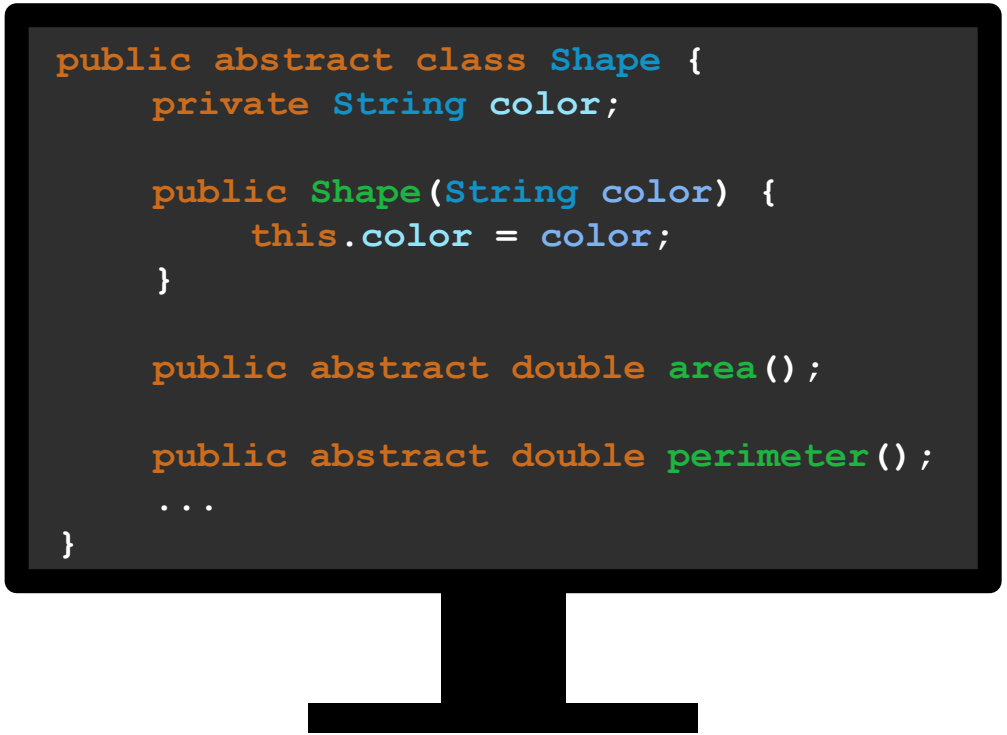
Extra Credit (Requires a bit of research):

February 02 1991 8:45am ➡ **Vadodara**

Interfaces

Abstract Class

- An **abstract class**
  - defined with the modifier ***abstract***
  - can contain ***abstract methods***
  - doesn't implement inherited abstract methods
  - Can't create instance of abstract class
- **Abstract method** - a method with keyword ***abstract***, ends with a semicolon instead o f method body
  - Private and static methods can't be ***abstract***.



```
public abstract class Shape {
    private String color;

    public Shape(String color) {
        this.color = color;
    }

    public abstract double area();

    public abstract double perimeter();
    ...
}
```

Interface

- **Interface** is a contract
- Contains **method signatures** (methods without implementation) and **static constants** (final static)
- Can only be implemented by classes and extended by other interfaces
- Used as a “template” for how a class should be structured