# 

# 

# 

# 

# 

***M&M Weather***

***A comprehensive developer’s guide***

Meir Manevich, Munkaran Singh

5 December 20

# 

Table of Contents

[**1. Introduction 3**](#_oshlopem2fir)

[1.1. Project Overview 3](#_483cec54yequ)

[**2. How to Run the Code 3**](#_2p0r8if50ogs)

[2.1. Requirements 3](#_3ghxcy1quxyt)

[2.2. Visual Studio Code Step-by-Step 3](#_nq9g7hav827u)

[**3. Design 3**](#_5qf9ohaaf5qo)

[3.1. Project Structure 3](#_3ii5os5249gs)

[**4. Implementation 4**](#_n3y0qfe7afm7)

[4.1. WeatherApplicationController.java 4](#_l7y81aktfqo7)

[4.1.1. General Layout 4](#_r5diswcfn8hp)

[4.1.2. Functions 4](#_72ndvnyzokve)

[4.1.2.1. initialize() 4](#_2e9kblqh1b1l)

[4.1.2.2. handleSave() 5](#_nboyo1gwv893)

[4.1.2.3. handleSearch() 5](#_9j9w38nj8z0b)

[4.1.2.4. TheMainSearchFunction() 5](#_9o1i24acwkh5)

[4.1.2.5. setImage() 6](#_38xu4p58h3v8)

[4.1.2.6. showAlert() 6](#_b95i8wokfvbw)

[4.1.2.7. updateWeatherInfo() 6](#_i2lyq52valpb)

[4.1.2.8. Remaining Functions 6](#_t3bua64rlnoa)

[4.2. The weatherObject class 6](#_m5ie7zca0ikw)

[4.2.1. Attributes 6](#_tys6t7anb7w)

[4.2.2. Methods 7](#_wiqkfauhr2uo)

[4.2.2.1. parseTime() 7](#_a9b6ipn9kspi)

[4.2.2.2. parseTimeHourOnly() 7](#_t4xgiydca47j)

[4.2.2.3. parseDate() 7](#_kaqn06ev662r)

[4.2.2.4. parseDayOfWeek() 8](#_702vx9uwr7mm)

[4.2.2.5. parseWMO() 8](#_qw0anvc04zv5)

[4.2.2.6. parseWMOforImage() 8](#_wdpjfsg30ff9)

[4.2.2.6. getLocation() 9](#_gl9d17dfg2bh)

[4.2.2.7. getWeather() 9](#_yd3yuua8qosq)

[4.2.2.8. getAPIresponse() 10](#_4d781jtwsxuf)

[4.2.2.9. parseAPIresponse() 10](#_c1nwaw8se0yx)

[4.2.3. Constructors 11](#_i02s4sy68ath)

[4.2.3.1. weatherObject() 11](#_9ppi1c7jr2w3)

[4.2.3.1. weatherObject(String cityName) 11](#_j56lnz5ivxso)

# **1. Introduction**

## 1.1. Project Overview

* This program is a homemade weather desktop web application, designed for ease of use and artistic expression. A user can search any city or zip code and see the current, hourly, and daily weather, along with a complementary image to illustrate the current weather conditions.

# **2. How to Run the Code**

## 2.1. Requirements

* Java must be installed on the system, preferably version 22 or higher.
* Any IDE that supports Gradle will be able to run the code. Netbeans, Visual Studio Code, IntelliJ IDEA, and Eclipse all support Gradle natively. The program will run on any platform with Java, whether it is Windows, MacOS, or Linux.

## 2.2. Visual Studio Code Step-by-Step

* Using Visual Studio code, all that is needed is the Gradle for Java extension.
  + Wait for the Gradle to build the project, and ignore any notifications/error-looking things that pop up. When it is complete, there should be a list of tasks available to run in the Gradle extension sidebar.
  + Navigate to the Gradle extension sidebar (the elephant icon), click on “Tasks” under “gradleproject1”, then “application”, and then run the “run” task.

# **3. Design**

## 3.1. Project Structure

* There are 4 main files to this project:
  + *App.java*
    - This contains the main method from which a JavaFX desktop application is instantiated using the *WeatherApplication.fxml*.
  + *WeatherApplicationController.java*
    - This file handles user input, uses the *WeatherObject* class in the *WeatherObject.java* file to retrieve data, and then updates the frontend with all the new information.
  + *WeatherObject.java*
    - This is a class in which the user’s search term is sent off and weather data is retrieved, populating all of the attributes of the object that is created with the search term as its parameter.
  + *WeatherApplication.fxml*
    - This file is in the “resources” folder and contains all of the code for the frontend, and hands off control to *WeatherApplicationController.java* where the bulk of the backend happens.
* There is also a folder of images in the “resources” folder that contains the 8 background images.

# **4. Implementation**

## 4.1. *WeatherApplicationController.java*

### 4.1.1. General Layout

* First, we get all of the JavaFX elements from *WeatherApplication.fxml* using the @FXML syntax. For all of those elements, a function is created to either update its field or to implement functionality (like for a button). Then all of those functions are called at the appropriate times, like updating all of the fields after an API call, or saving a new search term to the saved list after the user hits the save button.
* All of the String variables that will be used later are created.
* An empty *weatherObject* is created, named “defaultt”, using the empty constructor, so the *handleSave()* function can use the *getLocation()* method to check if the location the user is trying to save is valid or not.

### 4.1.2. Functions

#### 4.1.2.1. *initialize()*

* This function is called by the *handleSave()* function after successfully adding a new item. The *handleSave()* function only adds the user’s search term to a list variable. This function sets the contents of that list to the visible saved list user interface.
* The function then listens for a mouse click on any of the saved search terms using *setOnMouseClicked()*, and *getClickCount()* to listen for specifically a double mouse click. Upon receiving a double mouse click, it sets the text of the search term that was clicked to a String variable and then passes it as the parameter to the *TheMainSearchFunction()* function.

#### 4.1.2.2. *handleSave()*

* This function is activated by the user clicking the save button, which creates an event that is sent as the input parameter to this function. A series of checks are performed.
  + The first condition checks if there is no input and sends an alert accordingly.
  + The second condition similarly checks if the search term isn’t valid using the *getLocation()* method from the *defaultt* object and sends an alert accordingly.
  + The third checks if the list already contains the current search term, and if not, adds it to the list.
  + The fourth condition checks if there are now more than 10 items in the list, and if so, removes the first one.

#### 4.1.2.3. *handleSearch()*

* This function calls the *TheMainSearchFunction()* using the text assigned to the *cityInput* JavaFX textfield as its parameter.

#### 4.1.2.4. *TheMainSearchFunction()*

* This function assigns its parameter text to a String variable, *cityNameInput*, which will be used throughout.
* First, we check if *cityNameInput* is empty or isn’t a valid location using the *getLocation* method of the *defaultt* object and send an alert accordingly and return right there. If the input is valid, the program will skip over the condition and continue.
* A new *weatherObject* is created, named *currentWeatherSearch*, using *cityNameInput* as its parameter. Due to constructor overloading, this will activate the other *weatherObject* constructor which gets all of the weather data.
* Using the attributes of *currentWeatherSearch*, the function assigns all the String variables their respective values from the attributes.
  + Note: For the daily weather, we start at index 7. This is due to past weather data, and will be explained later, along with the rest of the weatherObject class.
* To set the background image, an if statement checks whether it is day or night using the *isDayOrNightBinary* attribute of the *currentWeatherSearch* object. For both day and night, a Switch-Case statement is used to check the String assigned to the *currentWeatherConditionsforImage* variable and calls the *setImage()* function with the path to the corresponding image as its parameter.
* Finally, the *updateWeatherInfo()* function is called to update all of the JavaFX elements with the new data.

#### 4.1.2.5. *setImage()*

* This function creates an image variable using the image path passed in as its parameter for the image source.
* The *weatherImage* JavaFX element is updated with that image using the built-in ImageView *setImage()* function.

#### 4.1.2.6. *showAlert()*

* This function creates a pop-up alert to notify the user of something, using two String parameters to set the Title and Message of the alert, and then waits until the user clicks the “Ok” button to dismiss the alert.

#### 4.1.2.7. *updateWeatherInfo()*

* This function calls all the functions to update the JavaFX fields, updating the user interface, passing in the corresponding variables as their parameters.

#### 4.1.2.8. Remaining Functions

* The rest of the functions all set the content of the JavaFX fields to the value of their parameters, which are all called in the *updateWeatherInfo()* function.

## 4.2. *The weatherObject class*

### 4.2.1. Attributes

* Attributes are created to accept the format of data from the API- JSON objects, JSON arrays, Strings, long integers, and doubles.

### 4.2.2. Methods

#### 4.2.2.1. *parseTime()*

* This method accepts a date-time string as its parameter and returns just the time as its output.
* The input string must be in the format YYYY-MM-DDTHH:MM, where T splits the string between the date and the time, and time is given in 24 hour format.
* This method only deals with the time, so an array is created by splitting on the “T”, and then the *currentTime24hr* variable is set to the 2nd element of the array, which is the time.
* *currentTime24hr* is then split into hours and minutes on the colon. The minutes are good to go, but the hours variable is parsed into an integer to do some math.
  + The current 12 hour time is obtained by performing the modulus operation on the 24 hour time using 12.
  + “am” or “pm” is assigned to the *amORpm* variable if the 24 hour time divided by two is 0 or 1.
* The hours, minutes, and *amORpm* are all concatenated together and returned.
* This method is used to display the exact time of sunrise and sunset for each of the days in the daily weather section.

#### 4.2.2.2. *parseTimeHourOnly()*

* This method is identical to the *parseTime()* method except it returns only the current hour with *amORpm*, and not the minutes.
* This method is used to display the hourly time for the hourly weather section.

#### 4.2.2.3. *parseDate()*

* This method, although not used in *WeatherApplicationController.java* (other than a testing print statement), is the blueprint for the *parseDayOfWeek()* function.
* Like *parseTime()*, this method takes in a mixed date-time string separated by a “T” and splits it into an array, but this time only deals with the first element of the array, the date.
* The date string is then split on its dashes (“-”) further into three strings: *year*, *month*, and *day*.
* A hashmap is created to associate the numerical months with their English counterparts, assigning 01 to “January”, 02 to “February”, and so on.
* The month string is concatenated with the day and the year and returned to the caller.

#### 4.2.2.4. *parseDayOfWeek()*

* This method alters the *parseDate()* method to return the English representation of the day of the week.
* Following the algorithm set forth by <https://www.almanac.com/how-find-day-week>, the last two integers of the year are set to a variable and then another variable is set to ¼ of the last two digits of the current year.
* Two hashmaps are created, one for the Month Key and the other for the English representations of the days of the week, with Saturday being 0.
* An integer 0-6 is calculated by adding the number of the last two digits of the year to ¼ of that number, plus the day number minus 1, plus the Month Key, and then taking the sum of all of that divided by seven. The calculation looks like this:
  + (lastTwoOfYear+fourthOfYear+day-1+monthKey.get(month))%7;
* This number 0-6 is then used to get the English day of the week from the other hashmap and the string is returned to the caller.

#### 4.2.2.5. *parseWMO()*

* This method takes in a long integer as its parameter, which should be the WMO code of the current weather, and then creates an integer representation of that long.
* A hashmap is used to create key-value pairs between all of the possible WMO codes and their corresponding weather conditions.
* The current weather condition string is returned to the caller.

#### 4.2.2.6. *parseWMOforImage()*

* This method takes in a *parseWMO()* string output as its input parameter and returns one of four generalizations of the current weather condition.
* This method is used to display the correct current weather images.

#### 4.2.2.6. getLocation()

* This method takes in a city name or zip code as a String as its input parameter and returns a JSON object of the location data of that location, which includes its coordinates.
* First we replace all spaces with “+” symbols because that is how the Geocoding API formats it, and assign it to the *userCity* variable. Then the *userCityGeocodingURL* variable is set using the API link but broken up where the city should be and concatenated with the *userCity* variable.
* We then create a new *HttpURLConnection* called *getGeocodingResponse* using the *getAPIresponse()* method with the custom *userCityGeocodingURL* as its parameter, which will return an *HttpURLConnection* with that link.
* We then pass the *getGeocodingResponse* connection to the *parseAPIresponse()* method which returns a string representation of what is at the URL, and we assign the whole thing to a String variable, *geocodingResponse*.
* Although all of our data is now in a String variable, we know that it is formatted as a JSON, so we use the JSON library to parse *geocodingResponse* into a JSON object by creating a new JSON parser, *locJSONparser*, and using it to parse the *geocodingResponse* string into the *locJSONparserResults* Json object.
* If the location was valid, all of the data will be in an array called “results”, so we check if our JSON object *locJSONparserResults* contains “results.” If it does, then we return the first element of that array, which is our location data. If not, we return the entire JSON object, because an invalid location response doesn’t contain the “results” array, only an object with the generation time. Without this condition, it wouldn’t return anything at all. This way we can perform our checks in *WeatherApplicationController.java* to see whether or not the response of *getLocation()* contains “results” or not.

#### 4.2.2.7. getWeather()

* This method takes two parameters: the latitude and longitude of a given city. These two values are obtained from the data returned by the *getLocation()* method.
* The rest of this method is similar to the *getLocation()* method. A url string is created with its coordinates replaced with the parameter variables, an *HttpURLConnection* is created using the *getAPIresponse()* method with our url, and a string variable receives all of the data from the API via the *parseAPIresponse()* method.
  + Note: this url string has “&past\_days=7” at the end of it, which retrieves daily weather for the past 7 days in addition to the next 7 days. This increases the length of the daily weather variable arrays to 14, and the first 7 elements are the past 7 days, with the current day’s weather being the 8th element, at index 7.
* A JSON parser is created and a JSON object, *weatherJSONparserResults*, is populated using the response of that parser on our string of data from the url.
* This method returns the entire JSON object, *weatherJSONparserResults*, to the caller.

#### 4.2.2.8. getAPIresponse()

* This method takes in a url as its input parameter and creates a URL object from it, *APIurl*, using the java URI and URL classes.
* An *HttpURLConnection, responseConnection* is created casting the *openConnection()* of *APIurl* into an *HttpURLConnection*.
* *responseConnection* is then set to the “GET” request method to pull all of the data from the link, and it is returned in this state.

#### 4.2.2.9. parseAPIresponse()

* This method receives the connection opened in *getAPIresponse()* and takes all of the data coming from it.
* It creates a StringBuilder object, *apiResponseJSONdata*, to construct the final string representation of all of the JSON data.
* Then it creates a scanner for the input stream of what was passed in as the parameter, which should be *responseConnection* from *getAPIresponse()*.
* It then appends to *apiResponseJSONdata* each line of the incoming data from the API and then returns the completed string to the caller.

### 4.2.3. Constructors

#### 4.2.3.1. weatherObject()

* This is an empty constructor, used to access the *getLocation()* method prior to creating the main *weatherObject* in *WeatherApplicationController.java* to check whether or not the user entered a valid city or zip code.

#### 4.2.3.1. weatherObject(String cityName)

* This is the main constructor, accessible when using the *weatherObject* constructor with a String parameter.
* The constructor first uses *getLocation()* to get the JSON object in the “results” array and assigns it to the JSON object, *coordinates*.
* From there, the city name, state, country, latitude, and longitude are assigned to their corresponding attributes.
* A new JSON object, *weather*, is created, using the *getWeather()* method and the latitude and longitude variables as the parameters.
* From there, all of the attributes are filled out by accessing the data from *weather* and assigning them appropriately.
* With the class attributes filled out, the program in *WeatherApplicationController.java* can proceed to access all of the data necessary to fill out all of the frontend fields.