Portfolio 1

Blake Skaja and Chidike Ubani

Table of Contents

Introduction

Fantasy football is one of the biggest sources of gambling revenue every year in the United States. With the rise of online one week fantasy football websites such as DraftKings, which generates over 500 million dollars of revenue per year, we figured that there must be a way to use a computer to statistically generate the “ideal” team and hopefully win a piece of this huge money pot. DraftKings is very easy to play. You are given a salary of 50,000 and have to fill various roster spots with players with the goal being to have the highest scoring team. Each player has a different salary, with the “best” players costing up to 9,000 and the cheapest being 3,000.

The idea of this application is simple. There are numerous websites such as ESPN, CBSSportline, RotoWire and others that create predictions for how well each player will perform per week, based off of their matchup, past performance and other factors. Using an API provided by FantasyFootballNerd.com, we are able to get all of these predictions from these various websites through a json string. We are then able to go to DraftKings and download the .cvs file that contains the players and their salaries for the week. Using this data, the user can generate the “ideal” lineup and other features.

We used Java Swing as a quick and easy way to the user to interact with our data. We utilize the features of Swing through using JList, JoptionPane, JtabbedPane, JscrollPane, Jbutton and more. We also go above and beyond what we have learned in this course so far by interacting with the FantasyFootballNerd API, using a REST calls and building JSONObjects to be used by our data analysis operations.

This project demonstrates the highest level of learning in blooms taxonomy of synthesis by taking these different API’s and being able to build something new, useful and functional.

Interaction with Materials

As per assignment requirements, one of the key things that this portfolio had to demonstrate was our interaction with the materials that we had discussed and covered in class.

Of the core concepts that we had talked about in class, we decided to use WindowBuilder to create a simple yet fully functional Java Swing user interface. We used several of the features that were explained in the labs, including JtabbedPane, Jlist, JcomboBox, Jbuttons, JscrollPane, JoptionPane and also extending the Jframe class for a customized PlayerStats display.

We also created a few data structure classes that allow us to create Player and GameSchedule objects that can then be viewed, edited and mutated with through the use of the FantasyFootballApplication UI. Here are a few things about these classes that we deemed to be important.

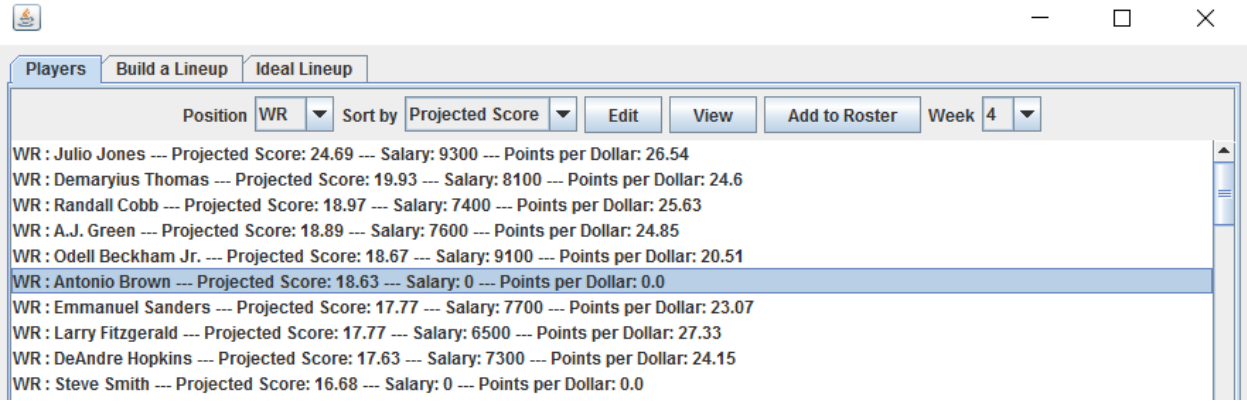
Player

The Player class was designed to essentially be a wrapper class that could be used in several different applications and extending past just football players. This implementation contains several public methods that allow a user to get any projection for any player, ranging from projected passing touchdowns to projected defensive sacks. By creating a simple and clean class, we allow for a consumer of this class to be able to use this player data however they want.

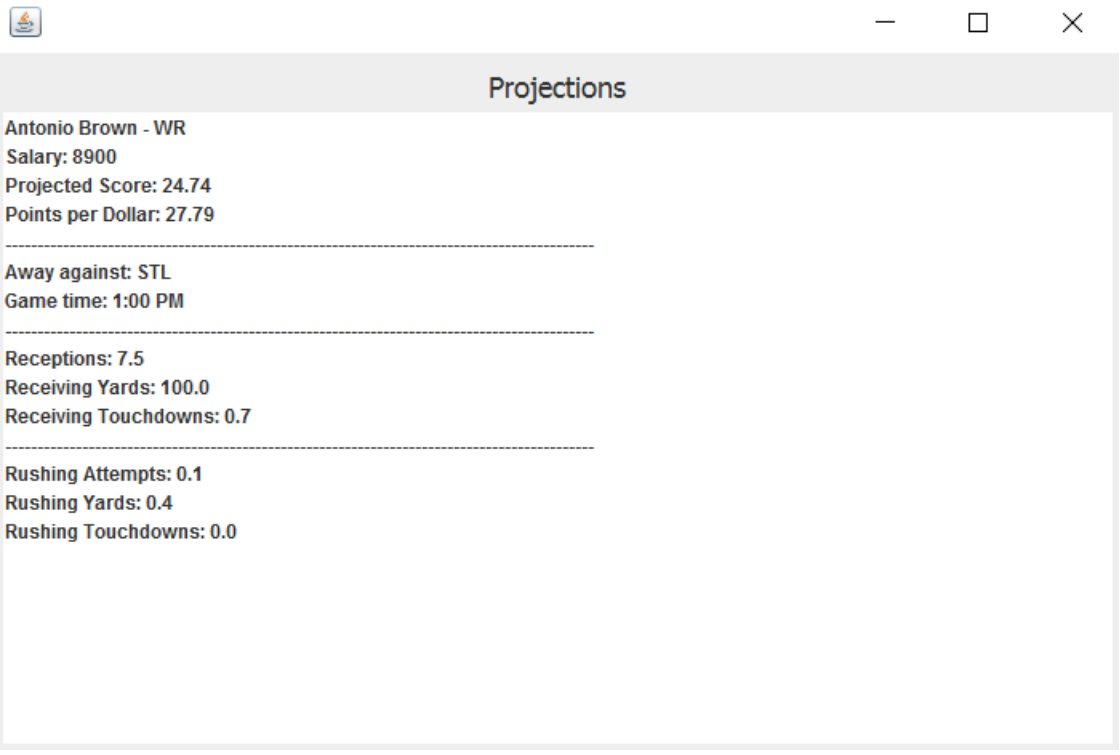
Our Player class allows us to create a player, give it projections data stored using a JSONObject, as well as generate a projected score based off of the DraftKings scoring system and the projections data. This can then later be used to generate full teams, display player stats and information and much more.

PlayerStats

In class we discussed using Jframe as a method of displaying data. In this project, we create an extension of the Jframe class to be used as a custom way for displaying our Player stats.



By pressing the View button in the header of our application, we create a new PlayerStats Object. This is populated with the data of the currently selected player. This data shows the projections for the player for the current selected week, the matchup for the team, and the time in which the game is being played.

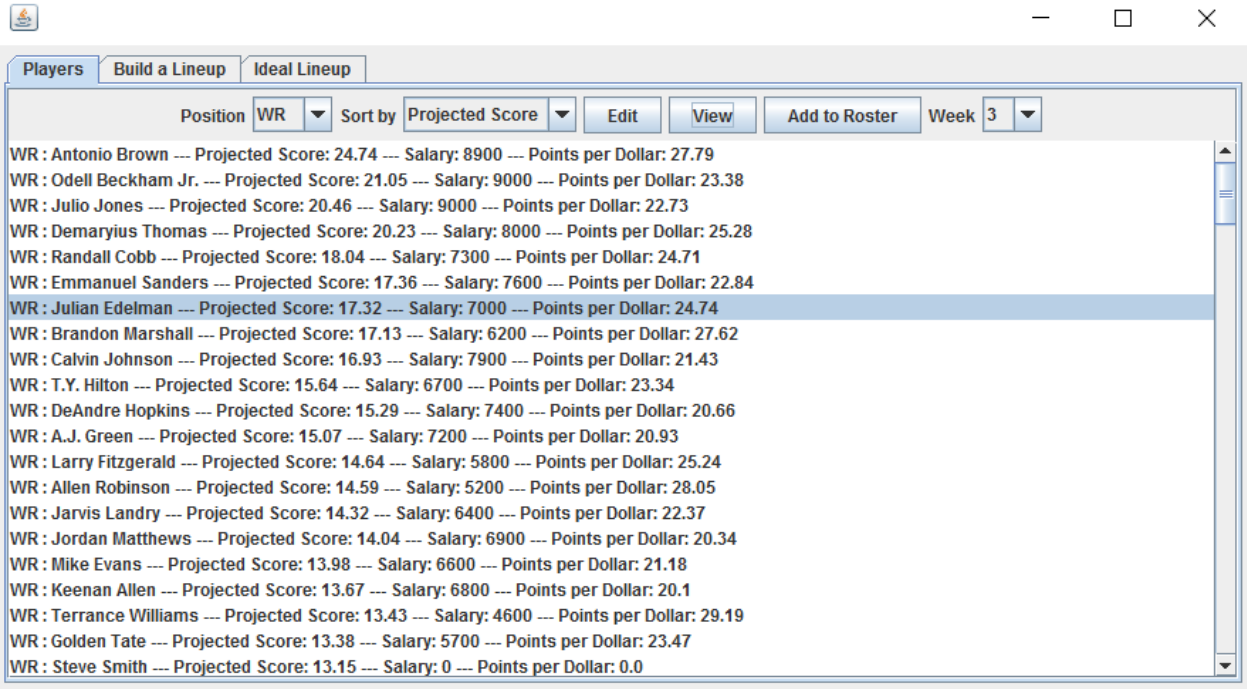


This allows for a user to open several different PlayerStats panels for side by side comparison of players. This is also a nice feature, as it allows for all of the players data to be seen in one, simple to read place. In a future iteration of this project, more information could be displayed, such as past game stats along with the current weekly projections.

FantasyFootballApplication

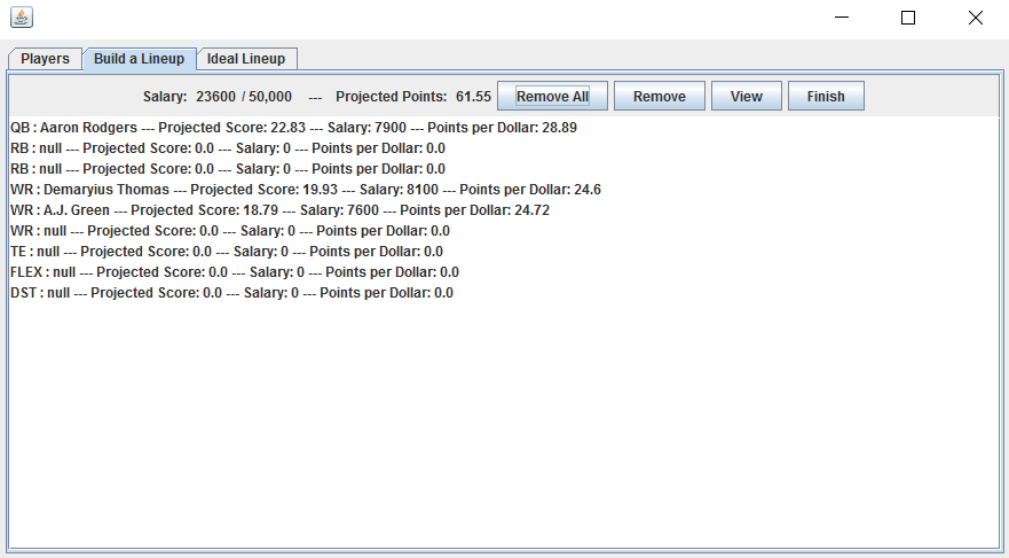
Since most of this project is based off of use of API's and backend calculations that allow us to generate money winning fantasy football line-ups, we decided that using WindowBuilder to quickly get a usable UI up would be the best solution. As I am not super artistic, being able to drag and drop button to make a simple yet functional UI is perfect for programmers like me.

There are three tabs in this application. The first tab called Players consists of a Jlist inside of a JscrollPane, as well as a header containing several Jbuttons, JcomboBoxes and some Jlabels.

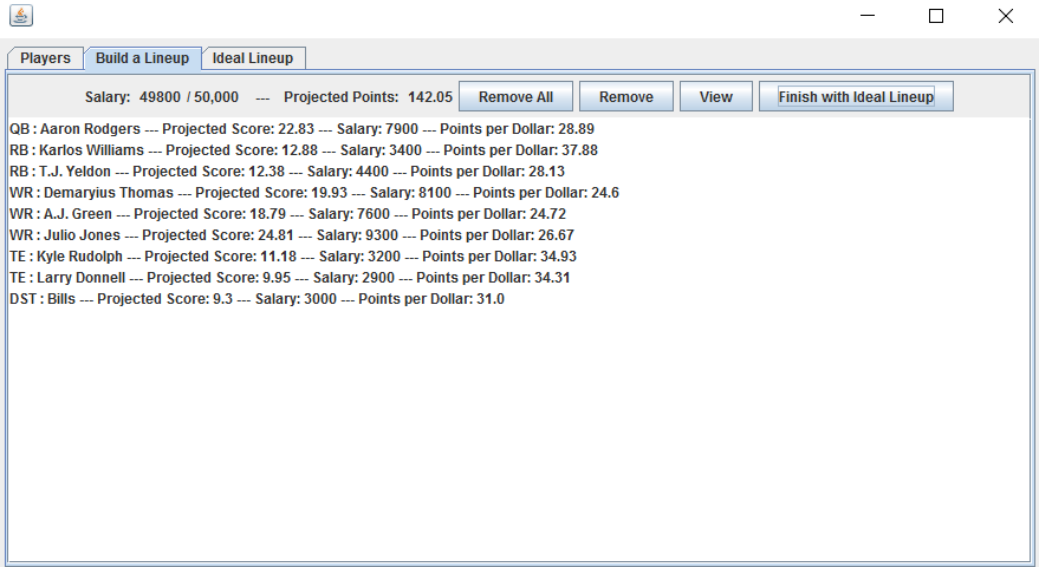


There are several JcomboBoxs that can be used to change the output of the Jlist. You can sort the players by position using the Position drop down selection. A user can also use the Sort By drop down selection to choose how the list should be sorted. A user can sort the list by Salary, by Projected Points or by Point per Dollar. This allows a user to quickly get information about the players. The Week drop down box allows the user to change the projections to display previous week's data. This can be used to get an idea of how a player has been performing previously.

In the header there are also several Jbuttons which can be used to interact with the Player data. The Edit button can be used to adjust the projected score of a player. Initially the players projected score is based off of the calculations made in the FantasyFootball generator. This score can be edited in the application however, if the user thinks that score is not an accurate prediction. This new score will be then used in generating line-ups. The View button will display more information about the player using a PlayerStats frame. The Add button will add the current selected player to the Build a Lineup list, which can be found on the next tab. This is one of the most useful and powerful abilities of this application and will be discussed further when talking about that tab.

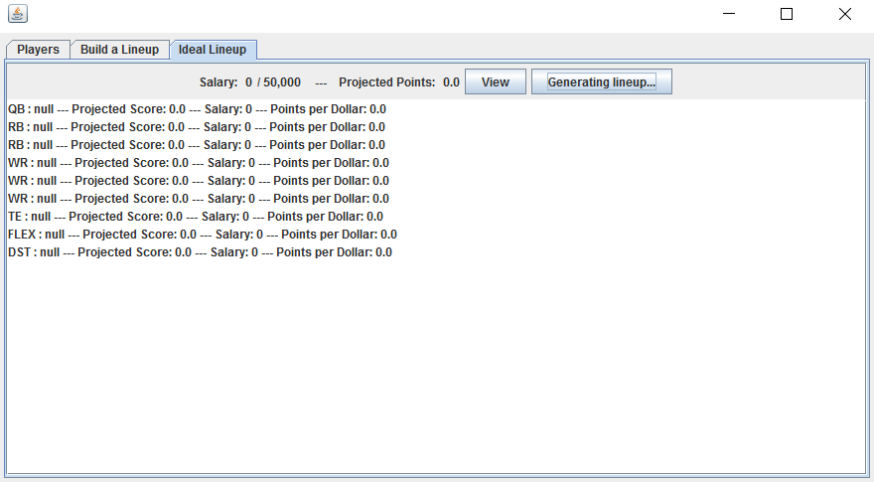


The second tab in the application is the Build a Lineup tab. This allows a user to add players from the Players tab and see what the salary and projected points of this team would be. Players can be removed from this lineup using the Remove button. The entire lineup can be cleared using the Remove All button. As before, the View button can be used to see advanced stats about the player. The last button, Finish, defines the essence of what this application is all about.

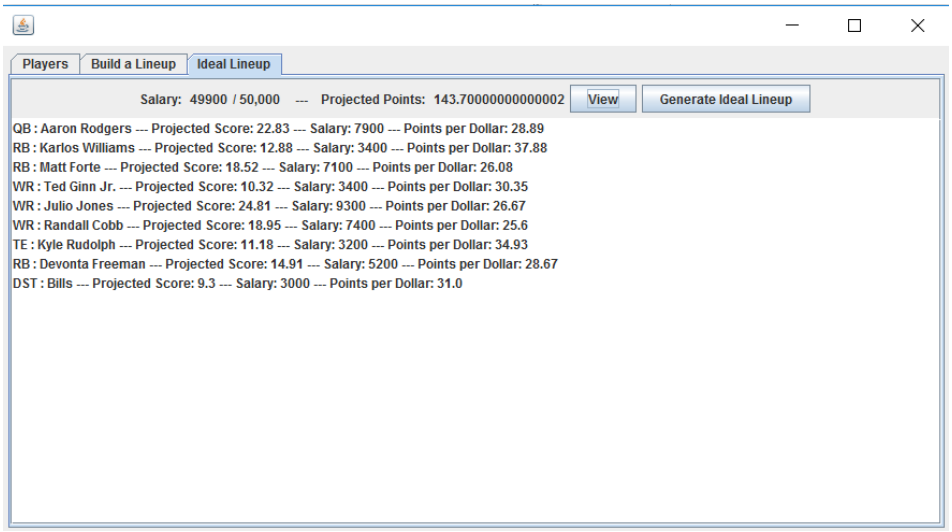


After pressing the Finish button, the rest of the lineup is completed. This lineup is the highest scoring lineup that can be generated while staying under the 50,000 dollar salary, containing the players that were previous added to the build a lineup tab. This is the power of this application. You can pick and choose as many players as you would like, and then have the application find the rest of the players to build you the best possible team.

The Ideal Lineup tab is the final tab in this application. This is similar to the Build a Lineup tab, but you are not allowed to add players to this list. This tab can generate the best possible team for you but does not accept any player selections.



This generation code can take a while to run, so we run this in its own thread so there the application does not lose any performance while the ideal lineup is being generated. We also use the SwingWorker doInBackground method to run this code in the background, which allows the user to continue using the application while the ideal lineup is being generated, instead of having the UI freeze up and wait for that chunk of code to be finished.



Exploration of More Complex Issues

While the first second focused on expanding on the concepts that we learned in class, this second is going to explore some complex topics that were used in this project that were not yet covered in this course. The use out outside API's are a big thing that were utilized in this project, which in my option is one of the most powerful things a good programmer can do. There is no need to rewrite a ton of code if someone else has already created a library that does that functionality for you. Through the use of FantasyFootballNerds and SimpleJSON API's, we were able to create a useable, robust project without doing outrageous amounts of work.

We will also talk about the FantasyFootball class that was created. This class uses these API's to do some powerful calculations. Some more advanced topics also include using SwingWorker to run background tasks and creating a custom thread to increase the performance of our application.

Fantasy Football Nerd API

http://www.fantasyfootballnerd.com/fantasy-football-api

The first API that was used in this application was the Fantasy Football Nerd API. The documentation for this API can be found at the link provided above. This API goes out to several fantasy football websites, including EPSN, CBSSportline, Rotowire and more. It gets the experts projections for how they believe that each player will perform in the upcoming weeks and creates a single projection based on all the different websites data. Data can be gathered from the API by using REST calls with the following URL format.

```
http://www.fantasyfootballnerd.com/service/ {SERVICE-NAME} / {FORMAT} / {API-KEY}
```

An example of this can be seen in the FantasyFootball class in the getProjectionsForPosition method.

```
public Map<String, Player> getProjectionsForPosition(String position, String week) throws IOException, ParseException {
    String url = "http://www.fantasyfootballnerd.com/service/weekly-projections/json/6s2562qvn9mg/" + position + "/" + week + "/";
    JSONArray allProjections = (JSONArray) runQuery(url).get("Projections");
    Map<String, Player> list = new HashMap<String, Player>();
    for(Object object : allProjections.toArray()) {
        JSONObject o = (JSONObject) object;
        list.put(o.get("displayName").toString(), new Player(o));
    }
    return list;
}
```

In this method, we create a String url that has the same format as shown above. In the method, we allow a consumer of this class to pass in a String position and a String week, which is concatenated into the url string.

This string is passed into the method runQuery(), which takes in a FantasyFootballNerd URL and uses a REST connection to get the data as a json String.

```
private JSONObject runQuery(String urlString) throws IOException, ParseException {
    URL url = new URL(urlString);
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();

    // Not sure why this is needed. It fixes the issues. Found it online.
    conn.setRequestProperty("User-Agent", "Mozilla/5.0 (Windows; U; Windows NT 6.1; en-GB;     rv:1.9.2.13) Gecko/20101203 Firefox/3.6.13 (.NET CLR 3.5.3
    if (conn.getResponseCode() != 200) {
            throw new IOException(conn.getResponseMessage());
    }

    BufferedReader rd = new BufferedReader(new InputStreamReader(conn.getInputStream()));
    StringBuilder sb = new StringBuilder();
    String line;
    while ((line = rd.readLine()) != null) {
        sb.append(line);
    }

    rd.close();

    conn.disconnect();

    JSONObject object = (JSONObject) parser.parse(sb.toString());
    return object;
}
```

This method takes in the urlString and creates a new HTTPUrlConnection. We then use a new InputStreamReader to get the connections input stream and pass that into a BufferedReader. Using a StringBuilder, we iterate through the BufferedReader and add the data into the StringBuilder. We close the BufferedReader, disconnect from the HTTP connection and then use another library called SimpleJson to parse the StringBuilder string and create a JSONObject.

Back in the getProjectionsForPosition method, we iterate through the new JSONArray that has been created and add the data into a HashMap for quick and easy retrieval. We use the players name as the key and create a new Player object based on the JSONObject data corresponding to that player. This allows us at any time to get the Players projections by using the Player's name as the key.

The FantasyFootballNerd API also provides data for game matchups, which allows us to determine what team each player is playing each week. This is great as matchups are a huge indicator in the success of fantasy football players. We can get the matchups through calling the getMatchups() method.

```
public ArrayList<GameSchedule> getMatchups() throws IOException, ParseException {
    String url = "http://www.fantasyfootballnerd.com/service/schedule/json/6s2562qvn9mg/";
    JSONArray allMatchups = (JSONArray) runQuery(url).get("Schedule");
    ArrayList<GameSchedule> gameSchedule = new ArrayList<GameSchedule>();

    for(Object object : allMatchups.toArray()) {
        JSONObject o = (JSONObject) object;
        gameSchedule.add(new GameSchedule(o));
    }
    return gameSchedule;
}
```

This again uses the runQuery() method to create a REST connection using the formatted url provided. Each of the matchups are then added to an ArrayList of GameSchedule objects, from which we can determine who each team plays each week.

Simple JSON API

https://code.google.com/p/json-simple/

Another API that is used in this project is the Simple.JSON API. There are several API's for Java that allow for building of JSON strings. We chose to use this one because of the simplicity and it provided the utilities that we needed in this project. The Simple JSON API provides several different classes, including JSONObject, JSONArray and JSONParser.

The JSONParser object is used in the FantasyFootball class in order to parse the string that is gathered from the REST calls to the FantasyFootballNerds API.

```
public class FantasyFootball {

    Map<String, Player> quarterbackList;
    Map<String, Player> runningbackList;
    Map<String, Player> recieverList;
    Map<String, Player> tightendList;
    Map<String, Player> flexList;
    Map<String, Player> defenseList;
    JSONParser parser;

    /**
```

Using this JSONParser, we are able to build a JSONObject that can be read like a map. JSON objects are really awesome, as they allow you to pass large amounts of data through a simple string. Using the JSONParser, we can build a JSONObject out of this String, which allows us to query the properties just like a HashMap.

```
/**
 * A player object for holding all the player related data
 * @param projections
 */
public Player(JSONObject projections) {
    projectionsData = projections;
    name = projections.get("displayName").toString().trim();
    teamName = projectionsData.get("team").toString();
    if (projectionsData.get("position").toString().equals("DEF")) {
        position = "DST";
        String[] splitList = name.split(" ");
        name = splitList[splitList.length - 1].trim();
    } else {
        position = projectionsData.get("position").toString();
    }
    projectedScore = Math.floor(calculateProjectedScore() * 100) / 100;
}
```

In the Player class, we access the properties.get("displayName") in order to get the name of the player. We can also use similar calls in the getter methods of that class in order to retrieve projections for each player.

```
public double getFieldGoalMadeProjection() {
    return Double.parseDouble(projectionsData.get("fg").toString());
}

public double getDefSafetyProjection() {
    return Double.parseDouble(projectionsData.get("defSafety").toString());
}

public double getFumblesLostProjection() {
    return Double.parseDouble(projectionsData.get("fumblesLost").toString());
}

public double getDefensiveTouchdownProjection() {
    return Double.parseDouble(projectionsData.get("defTD").toString());
}

public double getDefensivePointsAgainstProjection() {
    return Double.parseDouble(projectionsData.get("defPA").toString());
}

public double getDefensiveInterceptionsProjection() {
    return Double.parseDouble(projectionsData.get("defInt").toString());
}

public double getDefensiveYardsAllowedProjection() {
    return Double.parseDouble(projectionsData.get("defYdsAllowed").toString());
}
```

By wrapping these inside of methods, we are provided with a robust API that anyone can use to parse and get information from a Player object.

FantasyFootball

This is where the heart and soul of this project lives. In the previous sections, we have discussed how different API's were used to get the player data. But there is more to this class besides using API's. This class contains the players lists for each position and also the logical to generate lineups. We are able to map the DraftKings data from a .cvs file using a simple Scanner and finding the players in the HashMap.

```
/**
 * Maps the current DraftKings salaries to the players
 * @param week
 *  The week of the DraftKings salaries to used. Must be saved in the project folder as a .csv file with the format DKSalaries$week$.csv.
 */
private void assignSalariesToPlayers(String week) {
    File file = new File("DKSalaries" + week + ".csv");
    Scanner reader;
    try {
        reader = new Scanner(file);
    } catch (FileNotFoundException e) {
        return;
    }

    reader.nextLine();
```

After the players have been given their salaries for the week, the user can then use getIdealLineup to generate the highest scoring lineup. This method runs through all permutations of lineups using the top 14 QB's, top 25 RB's and wide WR's and the top 10 TE's: sorted by points per dollar.

```
/**
 * Generates the highest scoring possible line-up
 * @param selections
 * The players that you forsure want on your team
 * @return
 * The ideal line-up
 */
public ArrayList<Player> getIdealLineup(ArrayList<Player> selections) {
    ArrayList<Player> fullQBList = sortByPointsPerDollar(quarterbackList);
    List<Player> partQBList = fullQBList.subList(0, 14);

    ArrayList<Player> fullRBList = sortByPointsPerDollar(runningbackList);
    List<Player> partRBList = fullRBList.subList(0, 25);

    ArrayList<Player> fullWRList = sortByPointsPerDollar(recieverList);
    List<Player> partWRList = fullWRList.subList(0, 25);

    ArrayList<Player> fullTEList = sortByPointsPerDollar(tightendList);
    List<Player> partTEList = fullTEList.subList(0, 10);

    ArrayList<Player> fullFlexList = sortByPointsPerDollar(flexList);
    List<Player> partFlexList =  fullFlexList.subList(0, 25);

    List<Player> dstList = sortByProjectedPoints(defenseList);

    GetIdealLineupThread thread = new GetIdealLineupThread(partQBList, partRBList, partWRList, partTEList, partFlexList, dstList, selections);
    thread.run();

    return thread.idealLineup;
}
```

These lists are passed into the class called GetIdealLineupThread which runs the lineup generation algorithm in a separate thread in order to keep the application running fast.

GetIdealLineupThread has a lot of code, so I am not going to place that in the document here. That code can be found at the end of this document under the Code section. In this chuck of code, all possible lineups are generated, and the maximum scoring lineup that meets the salary cap is returned.

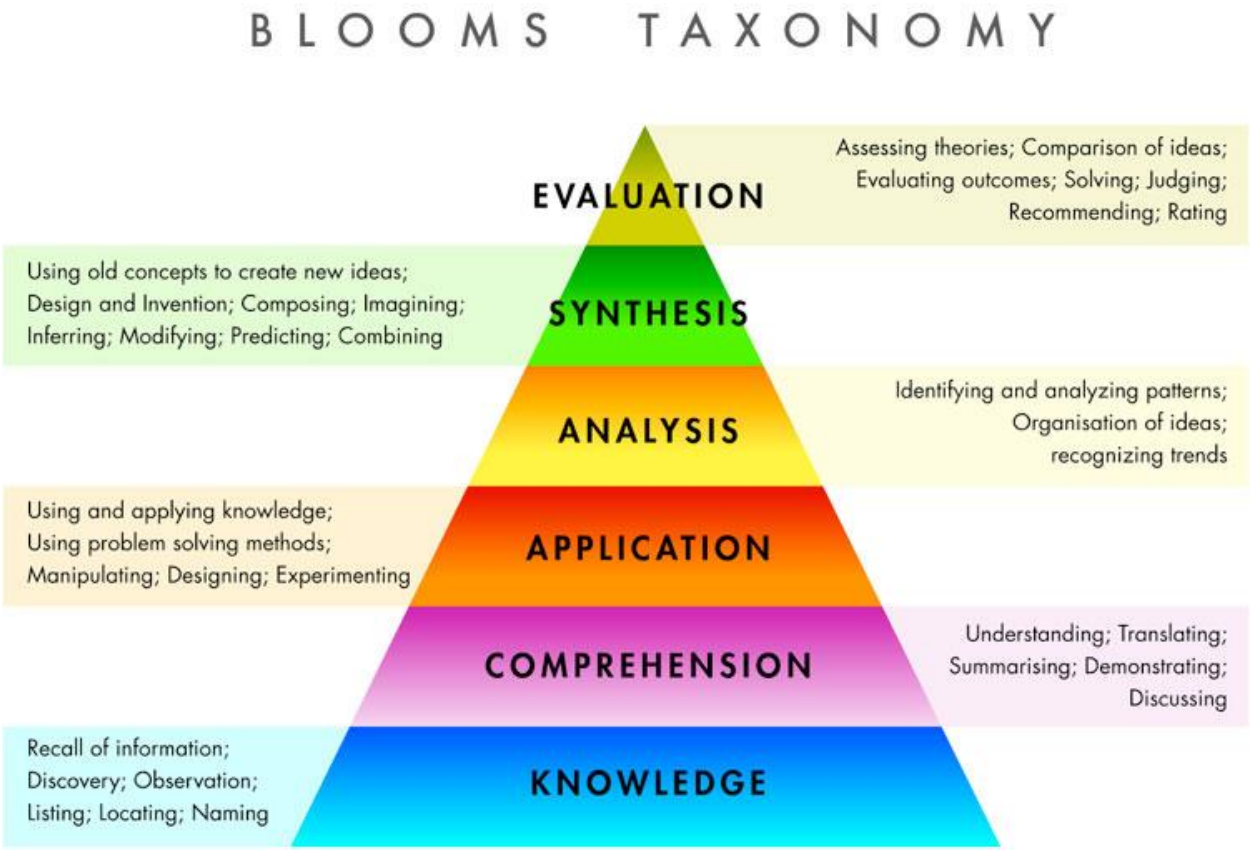
SwingWorker

This is a fairly minor new concept that was used, but never the less something that makes this application run much smother. Using the doInBackground() method of the SwingWorker, we are able to run the generate ideal lineup thread and also still navigate throughout the application without having it freeze up. Since generating the ideal lineup can take as long as 5 minutes, this allows a user to generate the lineup and continue to look at player data while the lineup is being created.

```
buildLineupFinishButton = new JButton("Finish");
buildLineupFinishButton.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent arg0) {
        SwingWorker worker = new SwingWorker() {
            @Override
            protected Object doInBackground() throws Exception {
                buildLineupFinishButton.setText("Generating lineup...");
                ArrayList<Player> idealLineup = football.getIdealLineup(choosenPlayers);
                choosenPlayers.clear();
                choosenPlayers.addAll(idealLineup);
                buildLineupList.setListData(idealLineup.toArray());
                updateLabels(buildLineupProjectedPointsLabel, buildLineupSalaryLabel, idealLineup);
                buildLineupFinishButton.setText("Finish with Ideal Lineup");
                return null;
            }
        };
        worker.execute();

    }
});
```

Blooms Taxonomy



In our project, we focused mainly on the second highest level of Blooms Taxonomy which is synthesis. Synthesis is the idea of taking concepts that were already known and using them to create a new product. This is evident in the Fantasy Football application that we created. We were able to take old ideas such as threading, Java Swing, Window builder and incorporate them with some new concepts such as using API's, JSONObjects and SwingWorker in order to create something new. These concepts can be seen throughout our code base and throughout the first portions of this document.

There is also evidence of analysis and application inside of our portfolio one project. We use analysis through organizing our code into smart, object oriented based code. This allow for reusability of code, as our classes could be reused in other projects if need be. We also use application, as we found a problem that needed to be solved and we designed a solution that provided the required utility. I have already used this application to win money on DraftKings and will continue to expand on this concept in order to develop the most accurate predictions possible.

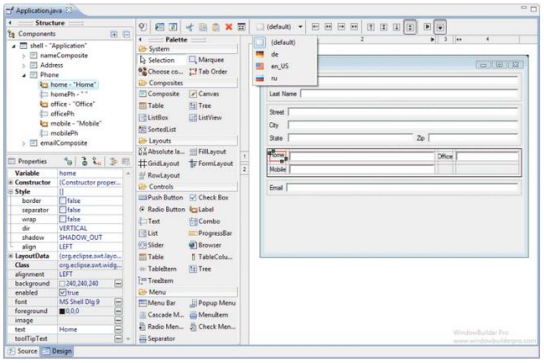
Remembering to focus on applying the principals of Bloom's Taxonomy in our portfolio was a great way to guide the work that we were doing. We were constantly focusing on creating something new and

innovative by synthesizing the concepts that we already knew as well as incorporating new ideas into our project. This is a useful skill that can be applied in all classes and professional settings. It is important to constantly be trying to innovate and create new things.

Examples of Blooms Taxonomy

We were able to incorporate blooms taxonomy in almost all of our project. Throughout the first 15 pages of this document is evidence of how we used Bloom's Taxonomy in this portfolio. Below we will highlight a few of the ways that we incorporated these ideas into our projects, but more evidence and examples are provided in the first 15 pages.

All throughout the FantasyFootballApplication we were able to use the concepts of WindowBuilder and Java Swing to create a new product that fit our needs. Our application did not need to be graphically intense, as it is simply a medium for displaying data to a user. This is at its core the idea of synthesis in Bloom's Taxonomy. We were able to take skills that we already had and create a new concept.



We were also able to use synthesis in using API's to create a fantasy football application that has practical use. Alone, the FantasyFootballNerds API has no use. By synthesizing our knowledge of football, coding, smart API use and object oriented code design, we were to create a useful application that people can use to hopefully win some money on DraftKings.

API Documentation

All services can be retrieved using simple REST calls. The URL construction is highlighted beneath each service. All service URLs will follow this pattern:

http://www.fantasyfootballnerd.com/service/ {SERVICE-NAME} / {FORMAT} / {API-KEY}

The example below will explain how to set the URL structure to retrieve all NFL teams.

The {SERVICE-NAME} is the specific service you want to retrieve.

For example, to retrieve all NFL teams, you would replace {SERVICE-NAME} with **nfl-teams**.

The {FORMAT} can have one of two options: **xml** or **json**

For example, to retrieve all NFL teams in JSON format, you would replace {FORMAT} with **json**.

The {API-KEY} is the API Key assigned to you. For testing purposes, you can also use the API Key of: **test**. Test calls will provide dummy data for you to develop against.

For example, you would replace {API-KEY} with **test** for testing purposes or with your actual API key for live data.

Example: http://www.fantasyfootballnerd.com/service/nfl-teams/json/test

We were able to use application and analysis by use of the JSON.Simple library. The idea of using json strings to store and pass around data is not a genius concept. After doing some reading and learning, it was fairly straight forward about how to using the JSONParser to parse a string and build a JSONObject out of it. This is the idea of using application, taking a concept, learning it and then being able to use it in a project. Our use of the JSON.Simple was a bit more on the analysis side however, as before using the json API, we first had to analyze and determine if this was the API that we wanted to use. By breaking down the technologies and finding a library that fit our needs, we were able to apply the concepts of analysis in this portfolio.