MANİSA
CELAL BAYAR
ÜNİVERSİTESİ

Data Structures

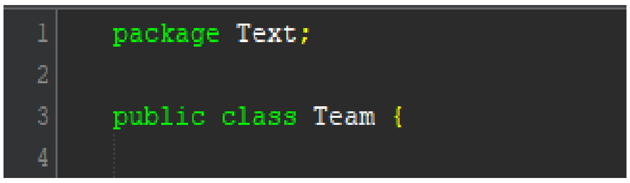
Football League Management System Project

220315040 -- Emine ÇETİN

220315082 -- Mustafa Furkan YILMAZ

TEAM CLASS

1. Class Definition

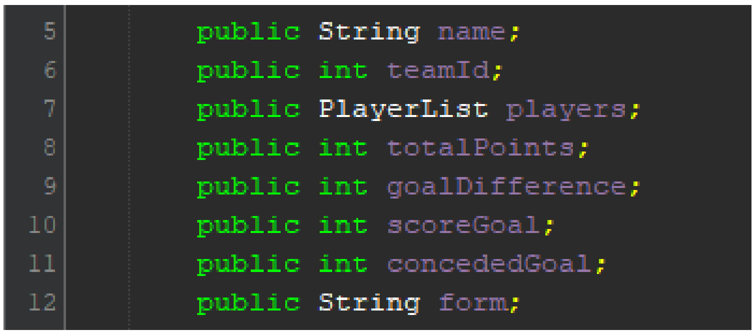


```java
package Text;

public class Team {

```

A class named Team is defined.

The statement package Text; indicates that this class is part of the Text package, helping to organize the project structure.

2. Fields



```java
        public String name;
        public int teamId;
        public PlayerList players;
        public int totalPoints;
        public int goalDifference;
        public int scoreGoal;
        public int concededGoal;
        public String form;
```

name: Stores the team's name.

teamId: Stores a unique identifier for the team.

players: Represents the list of players on the team (of type PlayerList).

totalPoints: Keeps track of the total points earned by the team.

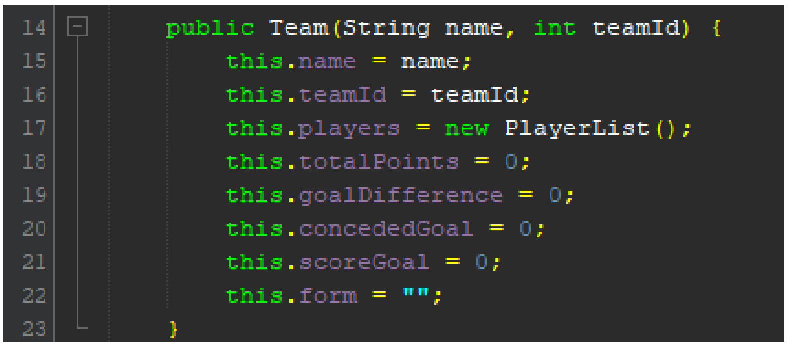
goalDifference: Stores the team's goal difference (but is dynamically calculated elsewhere).

scoreGoal: Tracks the total goals scored by the team.

concededGoal: Tracks the total goals conceded by the team.

form: Represents the team's performance or form (e.g., results of the last 5 matches).

3. Constructor



```java
    public Team(String name, int teamId) {
        this.name = name;
        this.teamId = teamId;
        this.players = new PlayerList();
        this.totalPoints = 0;
        this.goalDifference = 0;
        this.concededGoal = 0;
        this.scoreGoal = 0;
        this.form = "";
    }
```

When a Team object is created:

The team's name (name) and ID (teamId) are passed as parameters.

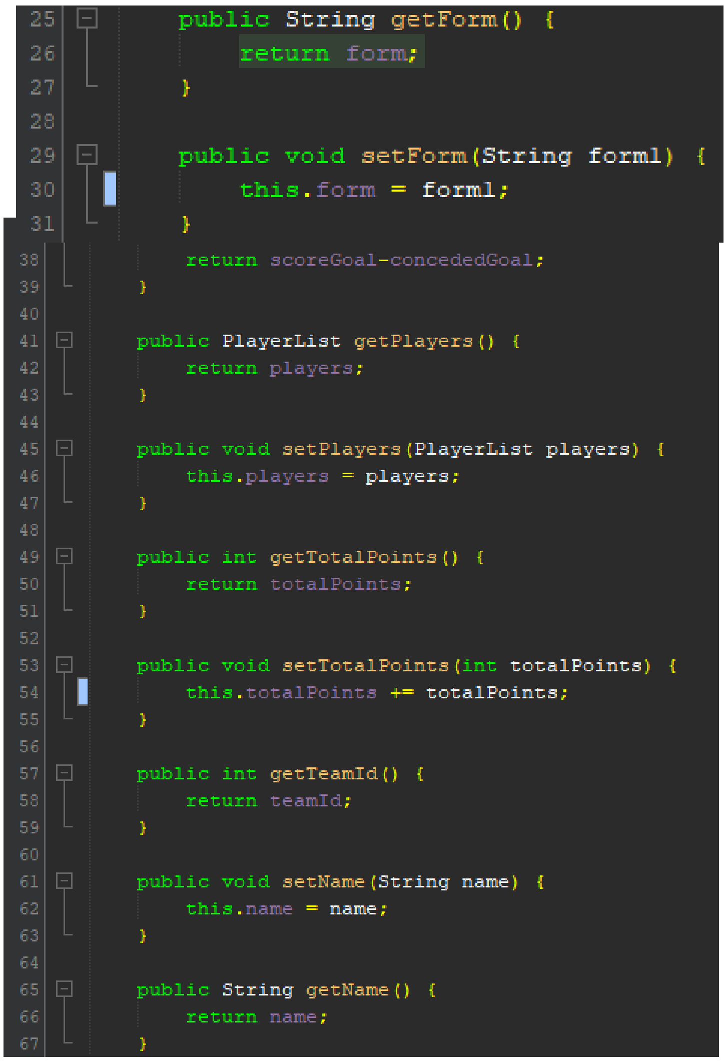
players are initialized as a new, empty player list (via the PlayerList class).

Other properties are initialized with default values:

Points and goals: 0

Form : An empty string ("").

4. Getter and Setter Method



```java
    public String getForm() {
        return form;
    }

    public void setForm(String form1) {
        this.form = form1;
    }
        return scoreGoal-concededGoal;
    }

    public PlayerList getPlayers() {
        return players;
    }

    public void setPlayers(PlayerList players) {
        this.players = players;
    }

    public int getTotalPoints() {
        return totalPoints;
    }

    public void setTotalPoints(int totalPoints) {
        this.totalPoints += totalPoints;
    }

    public int getTeamId() {
        return teamId;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
```

getForm: Returns the value of the form variable.

setForm: Intended to update the form,

getPlayers: Returns the list of players.

setPlayers: Sets a new player list for the team

getGoalDifference: Dynamically calculates the goal difference (scored goals - conceded goals).

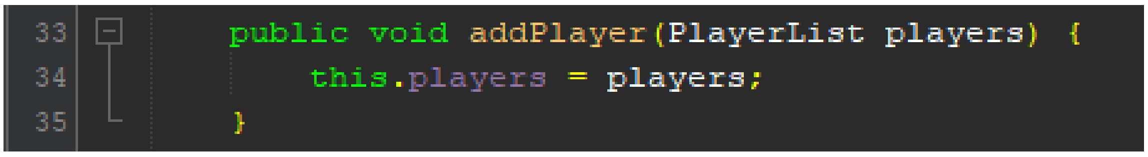
getTotalPoints: Returns the total points of the team.

setTotalPoints: Adds points to the total

getTeamId: Returns the team's unique identifier.

setName and getName: Used to set and get the team's name.

5. Additional Functionality



```java
    public void addPlayer(PlayerList players) {
        this.players = players;
    }
```

This method assigns a PlayerList type player list to the class's player’s field.

Purpose: To set or update the player list for the class using external input.

TEAMLIST CLASS

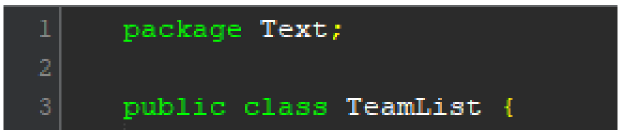
This TeamList class implements a singly linked list where:

Each node represents a team, connected via the next pointer.

Teams can be added, deleted, updated, and retrieved either by position or ID.

The list's size can be determined, and exceptions are handled for out-of-bound indices.

1. Class Definition

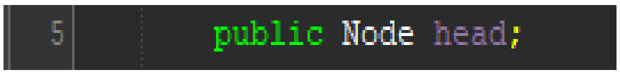


```java
package Text;

public class TeamList {
```

A class named TeamList is defined.

It is part of the Text package.

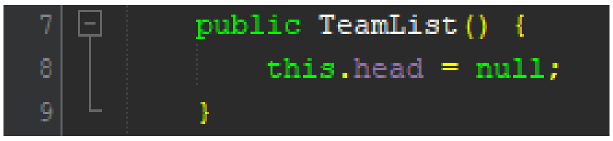
2. Fields



```java
    public Node head;
```

head: Represents the first node of the linked list. It serves as the starting point for traversing the list.

3. Constructor



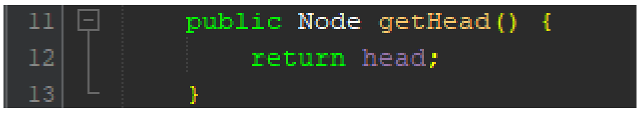
```java
    public TeamList() {
        this.head = null;
    }
```

The constructor initializes the linked list.

The head is set to null, indicating that the list is empty at the start.

4. Methods

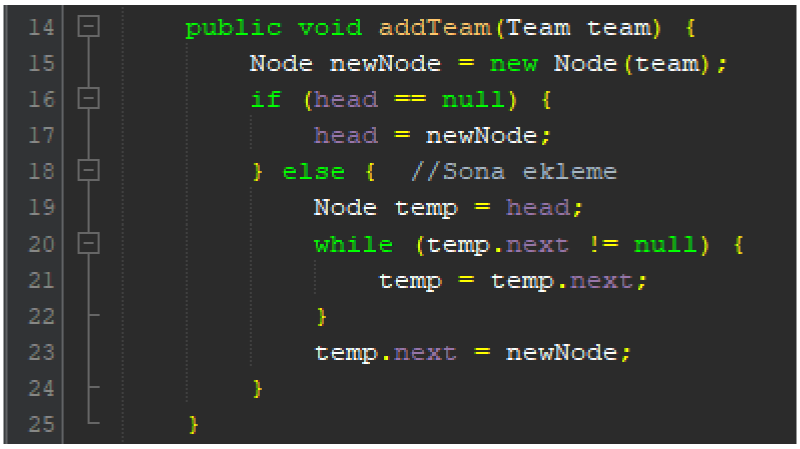
4.1 getHead()



```java
    public Node getHead() {
        return head;
    }
```

Returns the head of the list.

4.2 addTeam()



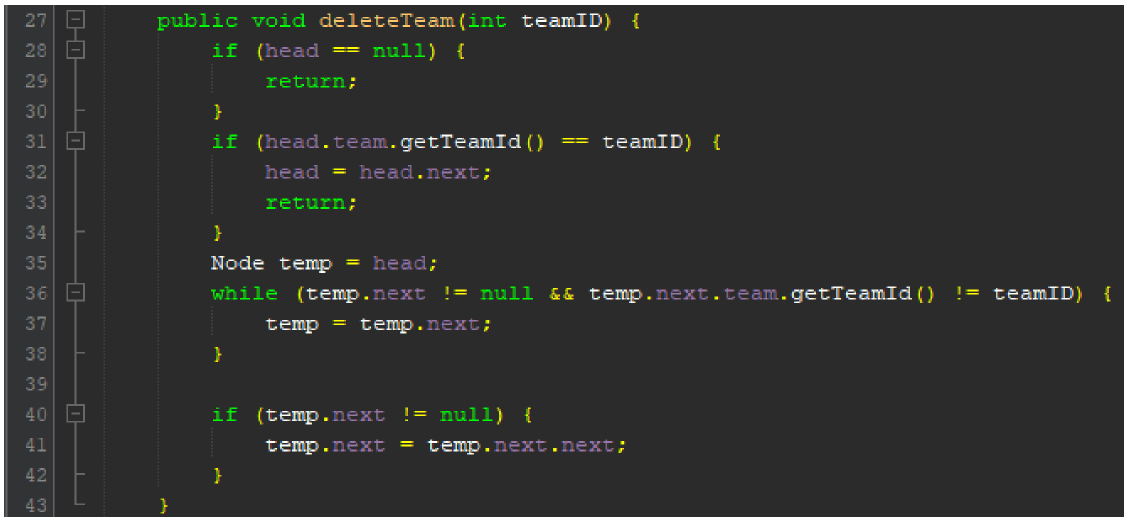
```java
    public void addTeam(Team team) {
        Node newNode = new Node(team);
        if (head == null) {
            head = newNode;
        } else {   //Sona ekleme
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
        }
    }
```

Adds a new Team object to the list.

If the list is empty (head == null), the new node becomes the head.

Otherwise, the method traverses to the end of the list and appends the new node.

4.3 deleteTeam()



```java
    public void deleteTeam(int teamID) {
        if (head == null) {
            return;
        }
        if (head.team.getTeamId() == teamID) {
            head = head.next;
            return;
        }
        Node temp = head;
        while (temp.next != null && temp.next.team.getTeamId() != teamID) {
            temp = temp.next;
        }

        if (temp.next != null) {
            temp.next = temp.next.next;
        }
    }
```

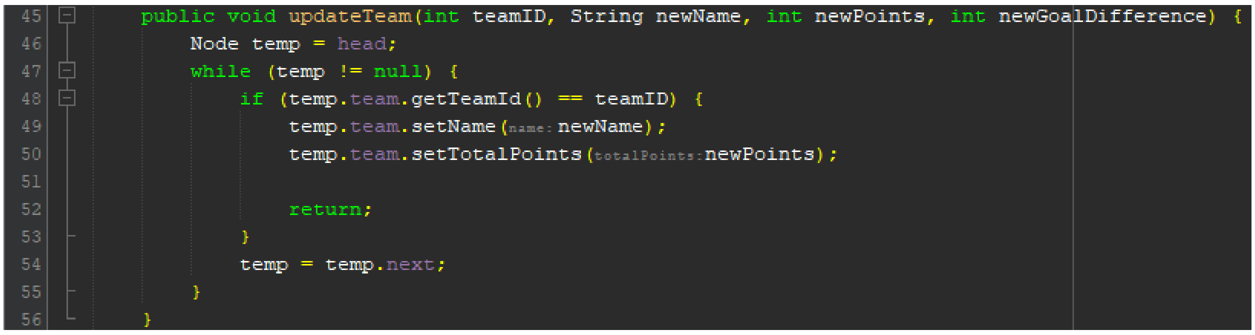
Deletes a Team from the list based on its teamID.

If the list is empty, nothing happens.

If the target team is the first node, the head is updated to the next node.

Otherwise, the method searches for the target node and updates the next pointer of the preceding node to skip the target.

4.4 updateTeam()

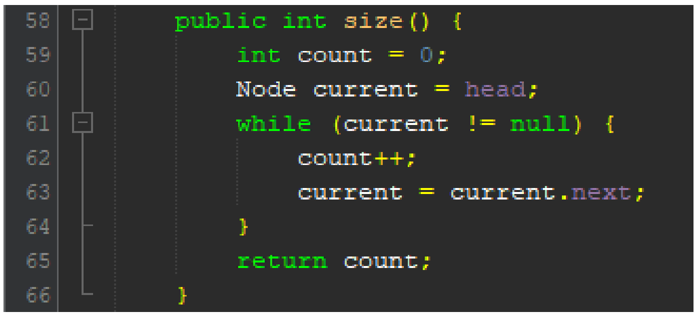


```java
    public void updateTeam(int teamID, String newName, int newPoints, int newGoalDifference) {
        Node temp = head;
        while (temp != null) {
            if (temp.team.getTeamId() == teamID) {
                temp.team.setName(name: newName);
                temp.team.setTotalPoints(totalPoints: newPoints);

                return;
            }
            temp = temp.next;
        }
    }
```

Updates the details of a team identified by its teamID.

Traverses the list until it finds the matching team, then updates its name and points.

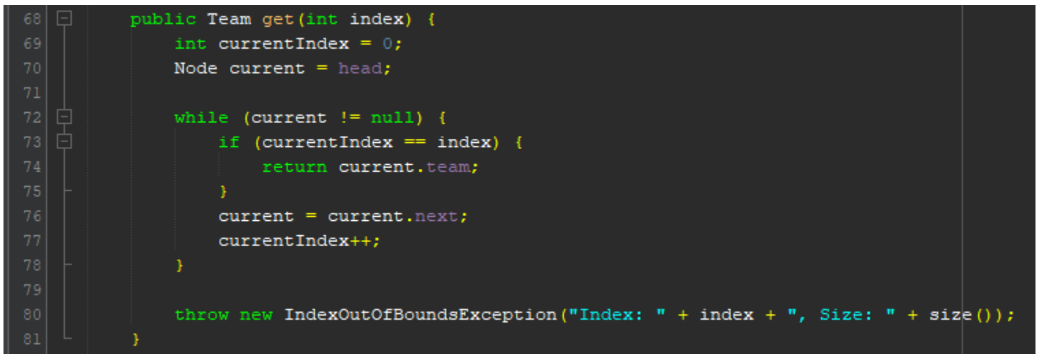
4.5 size()



```java
    public int size() {
        int count = 0;
        Node current = head;
        while (current != null) {
            count++;
            current = current.next;
        }
        return count;
    }
```

Returns the number of nodes (teams) in the list by iterating through it and counting.

4.6 get()

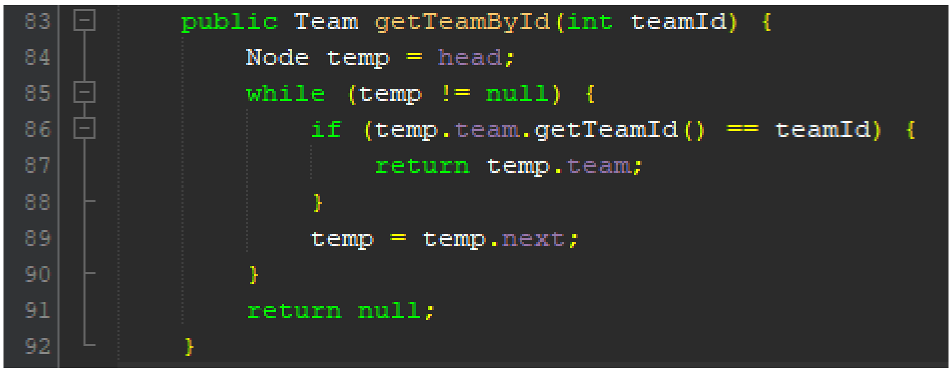


```java
    public Team get(int index) {
        int currentIndex = 0;
        Node current = head;

        while (current != null) {
            if (currentIndex == index) {
                return current.team;
            }
            current = current.next;
            currentIndex++;
        }

        throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size());
    }
```

Retrieves a Team based on its position (index) in the list.

Iterates through the list until the target index is found. If the index is out of bounds, an exception is thrown.

4.7 getTeamById()



```java
    public Team getTeamById(int teamId) {
        Node temp = head;
        while (temp != null) {
            if (temp.team.getTeamId() == teamId) {
                return temp.team;
            }
            temp = temp.next;
        }
        return null;
    }
```

Retrieves a Team based on its teamId.

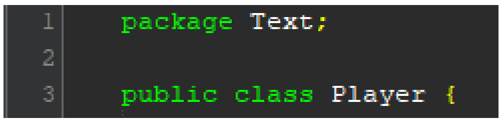
Searches the list for a matching teamId. If found, the corresponding Team is returned; otherwise, null is returned.

PLAYER CLASS

This class acts as a blueprint for creating player objects.

It includes fields for key player attributes and methods to set, update, or retrieve these attributes.

1. Class Definition

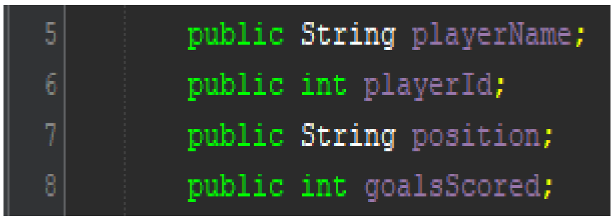


```java
package Text;

public class Player {
```

The Player class is part of the Text package.

Represents a player in a team.

2. Fields

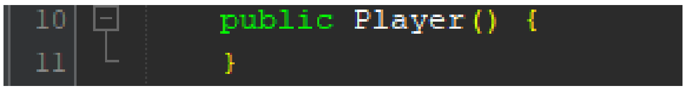


```java
    public String playerName;
    public int playerId;
    public String position;
    public int goalsScored;
```

playerName: Stores the name of the player.
playerId: Stores a unique identifier for the player.
position: Stores the player's position in the team (e.g., goalkeeper, forward, midfielder).
goalsScored: Tracks the total number of goals scored by the player.

3. Constructor

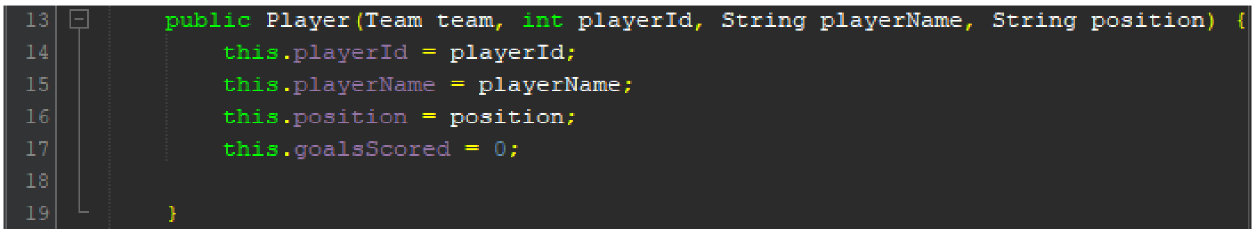
➔ **Default Constructor**



```java
    public Player() {
    }
```

Initializes a player object with no default values. Used when values are set later.

➔ **Parameterized Constructor**

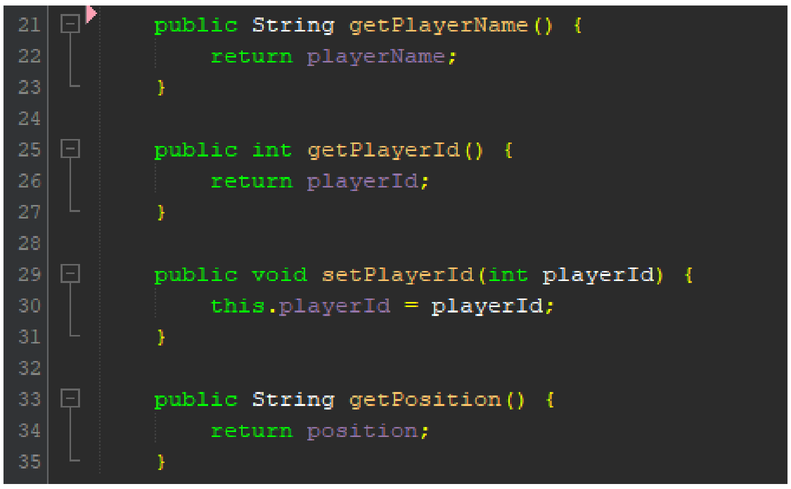


```java
    public Player(Team team, int playerId, String playerName, String position) {
        this.playerId = playerId;
        this.playerName = playerName;
        this.position = position;
        this.goalsScored = 0;

    }
```

Initializes a Player object with specific values:

team: (not used directly in this script, but the relationship is established externally).
playerId: Assigns a unique ID to the player.
playerName: Sets the player's name.
position: Assigns the player's position: Assigns the player's position.
goalsScored: Starts with 0 goals by default.

4. Getter and Setter Method



```java
    public String getPlayerName() {
        return playerName;
    }

    public int getPlayerId() {
        return playerId;
    }

    public void setPlayerId(int playerId) {
        this.playerId = playerId;
    }

    public String getPosition() {
        return position;
    }
```

getPlayerName: Returns the name of the player.

getPlayerId: Returns the unique ID of the player.

setPlayerId: Sets the player's unique ID to a new value.

getPosition: Returns the player's position in the team

PLAYERLIST CLASS

This class allows you to manage a list of players using a linked list data structure.

You can:

Add players using addPlayer().

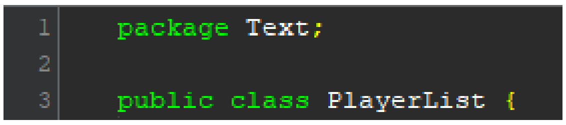
Remove players using remove().

Update player details using updatePlayer().

Count players with size().

Retrieve players by index using get().

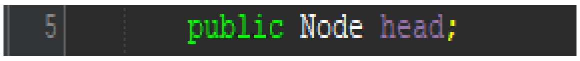
1. Class Definition



```java
package Text;

public class PlayerList {
```

The PlayerList class is part of the Text package.
This class manages a collection of Player objects.

2. Fields

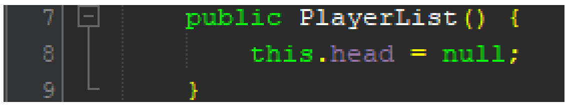


```java
    public Node head;
```

head: The first node in the list.

A Node is a data structure that holds a reference to a Player object and the next Node in the list.

3. Constructor

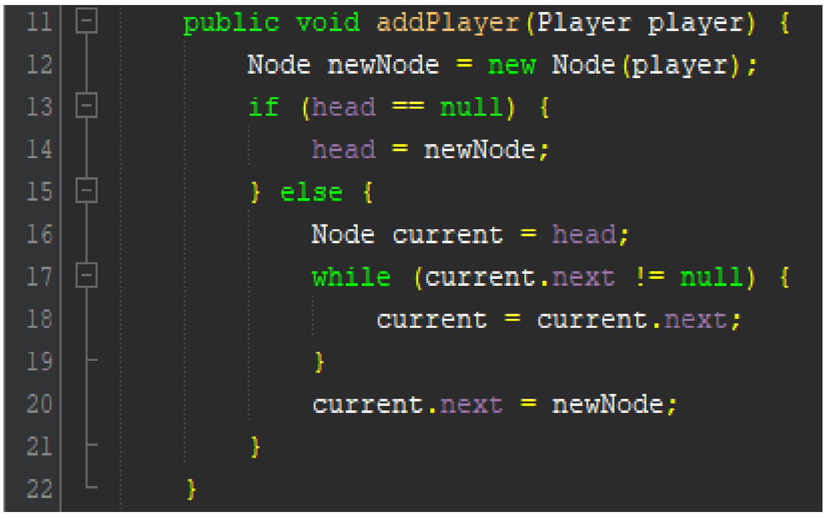


```java
    public PlayerList() {
        this.head = null;
    }
```

Initializes the list with no players. The head is set to null, meaning the list is empty at the start.

4. Methods

4.1 addPlayer()



```java
    public void addPlayer(Player player) {
        Node newNode = new Node(player);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
    }
```

Purpose: Adds a new player to the list.

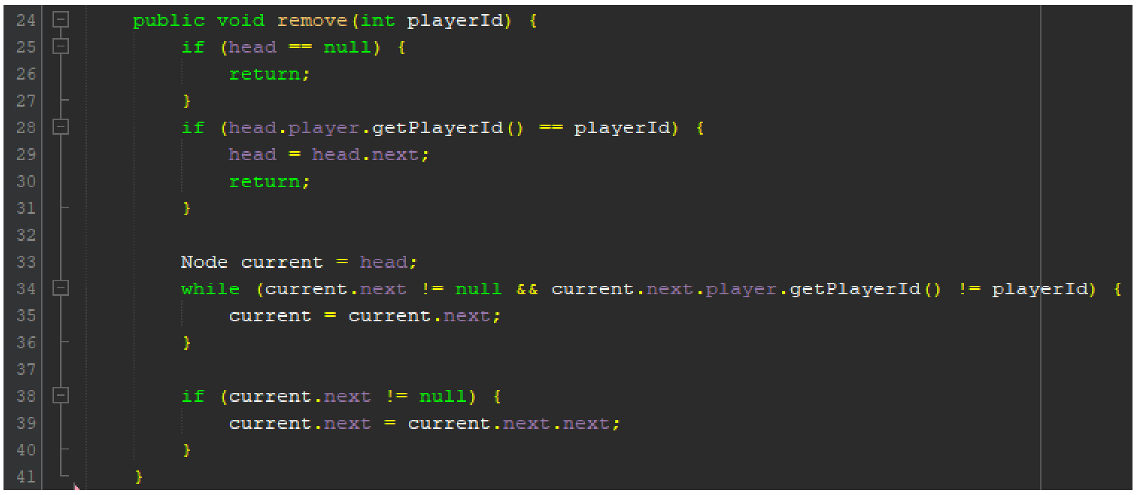
Steps:

A new node is created using the given Player object.

If the list is empty (head == null), the new node becomes the head of the list.

If the list already has players, it traverses to the last node and adds the new node at the end.

4.2 remove()



```java
    public void remove(int playerId) {
        if (head == null) {
            return;
        }
        if (head.player.getPlayerId() == playerId) {
            head = head.next;
            return;
        }

        Node current = head;
        while (current.next != null && current.next.player.getPlayerId() != playerId) {
            current = current.next;
        }

        if (current.next != null) {
            current.next = current.next.next;
        }
    }
```

Purpose: Removes a player from the list by playerId.

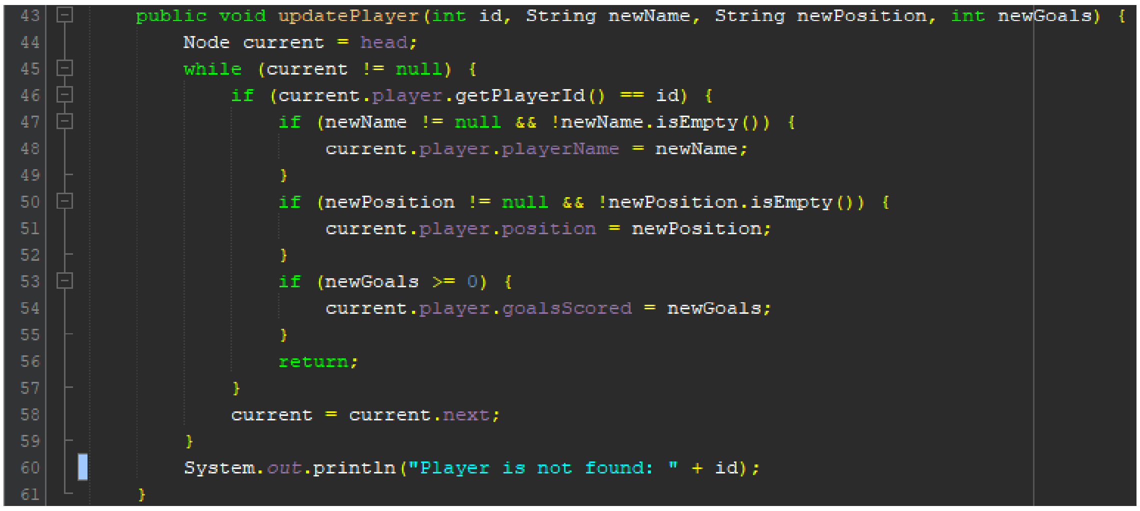
Steps:

If the list is empty, it does nothing.

If the player to be removed is the head node, the head pointer is updated to the next node.

If the player is not at the head, the list is traversed until the player is found, and the corresponding node is removed by adjusting the next pointer.

4.3 updatePlayer()



```java
    public void updatePlayer(int id, String newName, String newPosition, int newGoals) {
        Node current = head;
        while (current != null) {
            if (current.player.getPlayerId() == id) {
                if (newName != null && !newName.isEmpty()) {
                    current.player.playerName = newName;
                }
                if (newPosition != null && !newPosition.isEmpty()) {
                    current.player.position = newPosition;
                }
                if (newGoals >= 0) {
                    current.player.goalsScored = newGoals;
                }
                return;
            }
            current = current.next;
        }
        System.out.println("Player is not found: " + id);
    }
```

Purpose: Updates a player's information (name, position, or goals) by playerId.

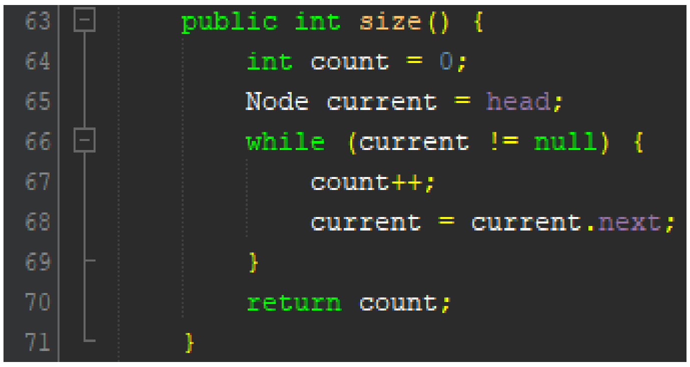
Steps:

Traverses the list and checks if the current player's playerId matches the given id.

If the player is found, the name, position, and goals are updated, if the new values are valid.

If the player is not found, a message is printed: "Player is not found: ".

4.4 size()



```java
public int size() {
    int count = 0;
    Node current = head;
    while (current != null) {
        count++;
        current = current.next;
    }
    return count;
}
```

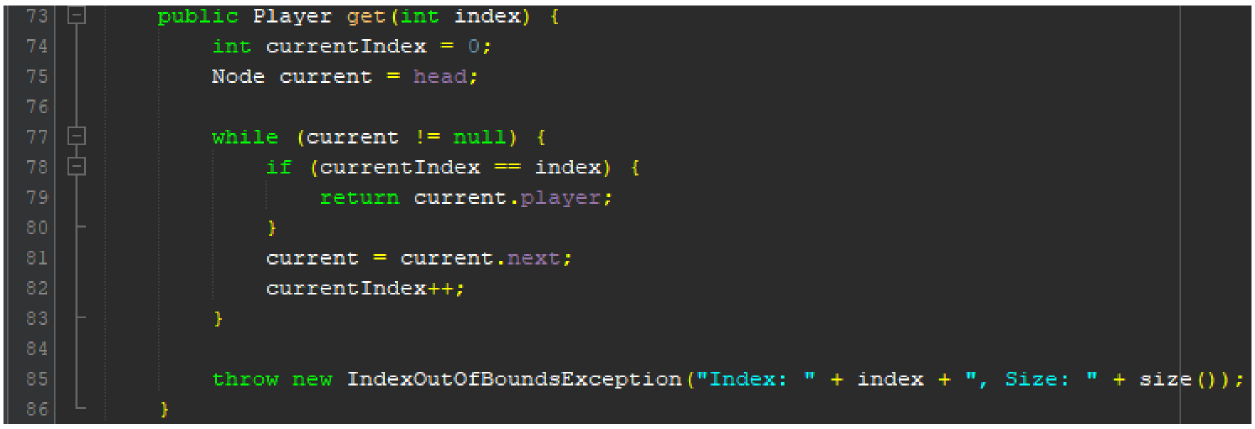
Purpose: Returns the number of players in the list.

Steps:

The list is traversed from the head to the end, incrementing the count for each node (i.e., player) encountered.

Finally, the size of the list (number of players) is returned.

4.5 get()



```java
public Player get(int index) {
    int currentIndex = 0;
    Node current = head;

    while (current != null) {
        if (currentIndex == index) {
            return current.player;
        }
        current = current.next;
        currentIndex++;
    }

    throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size());
}
```

Purpose: Retrieves a player by their index in the list.

Steps:

The list is traversed, and each player's index is compared to the given index.

If a match is found, the corresponding player is returned.

If the index is out of bounds (i.e., greater than or equal to the size of the list), an exception is thrown.

NODE CLASS

This class is designed for flexibility, allowing it to hold references to different types of objects (Player, Match, Team, or TeamList).

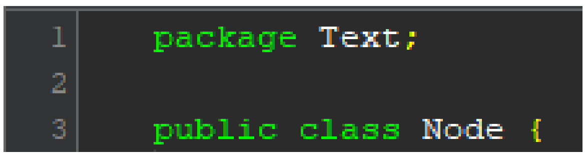
Usage in a Linked List:

The next field is used to connect nodes, forming a chain-like structure.

Depending on the context, the node can represent different entities in the application

(e.g., players, teams, matches).

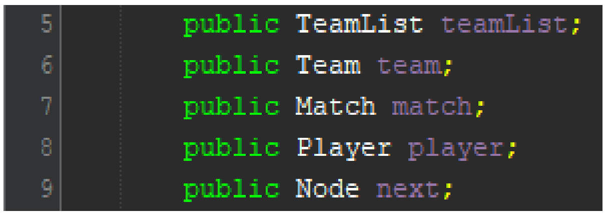
1. Class Definition



```java
package Text;

public class Node {
```

The Node class is part of the Text package.
This class represents a node that can hold different types of objects (TeamList, Team, Match, or Player) in a linked list structure.

2. Fields

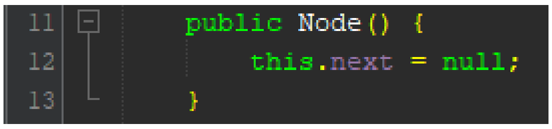


```java
    public TeamList teamList;
    public Team team;
    public Match match;
    public Player player;
    public Node next;
```

teamList: Holds a reference to a TeamList object.
team: Holds a reference to a Team object.
match: Holds a reference to a Match object.
player: Holds a reference to a Player object.
next: Points to the next node in the linked list.

3. Constructor

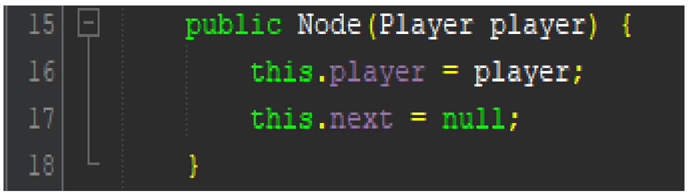
➔ Default Constructor



```java
    public Node() {
        this.next = null;
    }
```

Initializes an empty Node object.
Sets next to null, indicating this node doesn't point to another node yet.

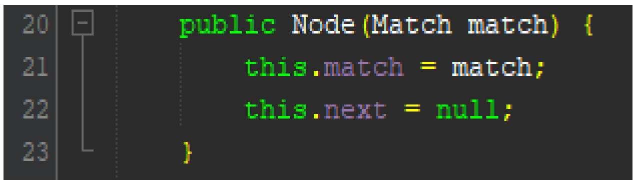
➔ Node(Player player) Constructor



```java
    public Node(Player player) {
        this.player = player;
        this.next = null;
    }
```

Initializes a Node object to hold a Player object.
Assigns the provided player to the player field.
Sets next to null

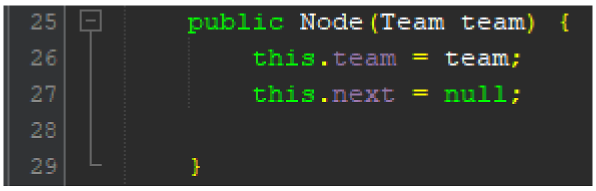
➔ Node(Match match) Constructor



```java
    public Node(Match match) {
        this.match = match;
        this.next = null;
    }
```

Initializes a Node object to hold a Match object.
Assigns the provided match to the match field.
Sets next to null.

➔ Node(Team team) Constructor



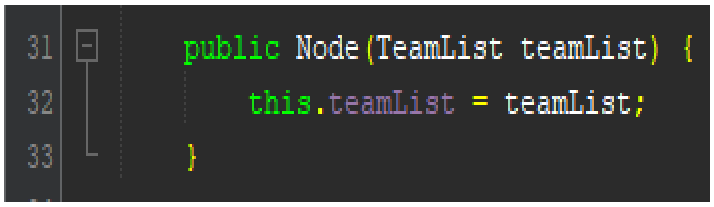
```java
    public Node(Team team) {
        this.team = team;
        this.next = null;

    }
```

Initializes a Node object to hold a Team object.

Assigns the provided team to the team field.

Sets next to null.

➔ Node(TeamList teamList) Constructor



```java
    public Node(TeamList teamList) {
        this.teamList = teamList;
    }
```

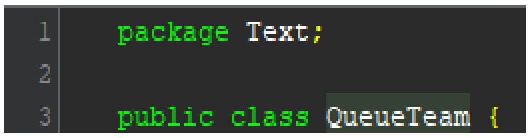
Initializes a Node object to hold a TeamList object.

Assigns the provided teamList to the teamList field.

Note: This constructor does not explicitly set next to null, but it will default to null if not assigned.

QueueTeam CLASS

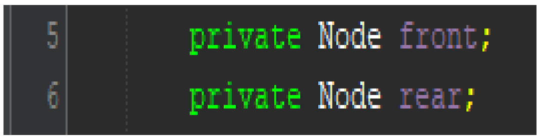
1. Class Definition



```java
package Text;

public class QueueTeam {
```

The QueueTeam class is part of the Text package.
It implements a queue data structure specifically for storing Match objects.

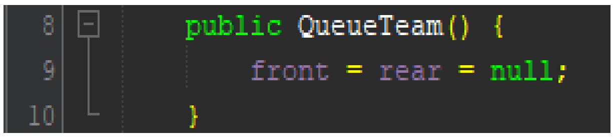
2. Fields



```java
        private Node front;
        private Node rear;
```

front: Points to the first (oldest) node in the queue.
rear: Points to the last (newest) node in the queue.
Both front and rear are initialized to null in the constructor.

3. Constructor

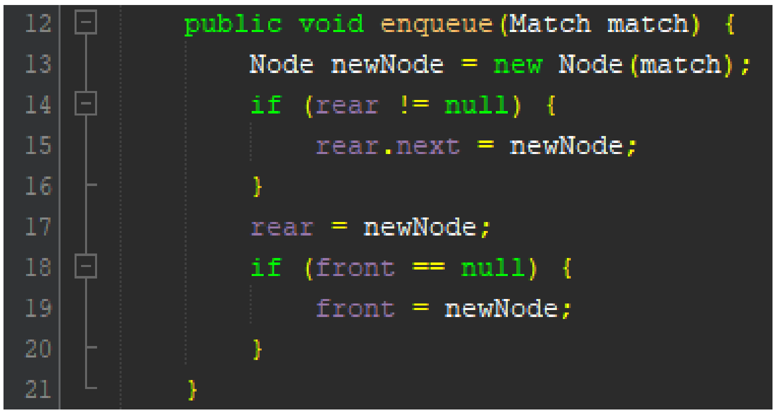


```java
    public QueueTeam() {
        front = rear = null;
    }
```

Initializes an empty queue.
Both front and rear are set to null, meaning the queue starts empty.

4. Methods

4.1 enqueue()



```java
    public void enqueue(Match match) {
        Node newNode = new Node(match);
        if (rear != null) {
            rear.next = newNode;
        }
        rear = newNode;
        if (front == null) {
            front = newNode;
        }
    }
```

Purpose: Adds a new Match to the end of the queue.

Steps:

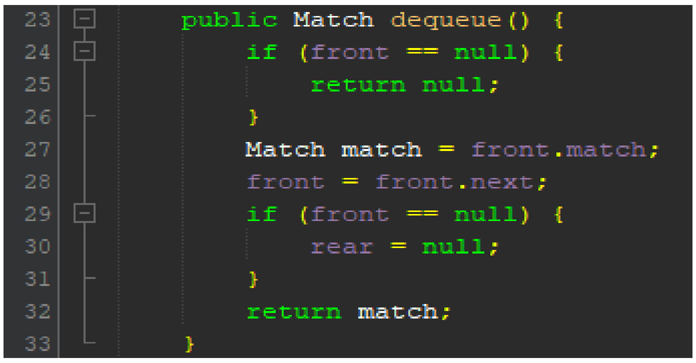
Create a new Node that holds the provided Match object.

If rear is not null, link the new node to the current rear node by setting rear.next to newNode.

Update rear to point to the new node.

If the queue is empty (front == null), set front to point to the new node as well (since it's now both the front and rear of the queue).

4.2 dequeue()



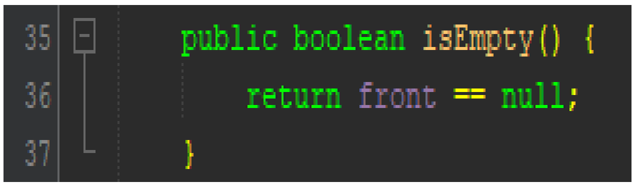
```java
public Match dequeue() {
    if (front == null) {
        return null;
    }
    Match match = front.match;
    front = front.next;
    if (front == null) {
        rear = null;
    }
    return match;
}
```

Purpose: Removes and returns the first Match in the queue.
Steps:
Check if the queue is empty (front == null). If so, return null.

Retrieve the Match object stored in the current front node.

Update front to point to the next node (front.next).
If the queue becomes empty after the removal (front == null), set rear to null as well.
Return the retrieved Match.

4.3 isEmpty()



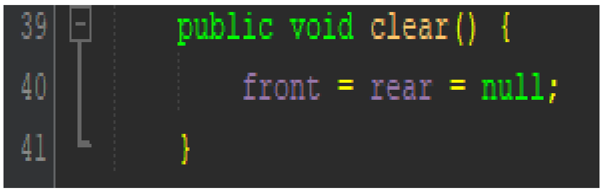
```java
public boolean isEmpty() {
    return front == null;
}
```

Purpose: Checks whether the queue is empty.

Returns:

true if front is null (i.e., no elements in the queue).
false otherwise.

4.4 clear()



```java
public void clear() {
    front = rear = null;
}
```

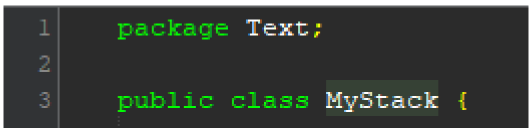
Purpose: Empties the queue.

Steps:

Sets both front and rear to null, effectively removing all nodes from the queue.

MyStack CLASS

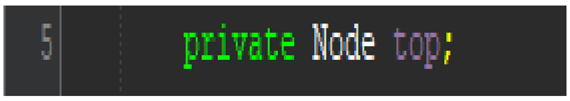
1. Class Definition



```java
package Text;

public class MyStack {
```

The MyStack class is part of the Text package.
It represents a stack data structure that stores Match object

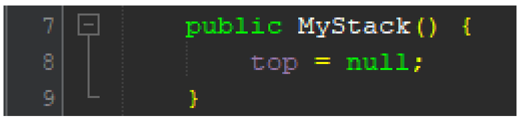
2. Fields



```java
    private Node top;
```

top: Points to the top element of the stack.
If the top is null, the stack is empty.

3. Constructor

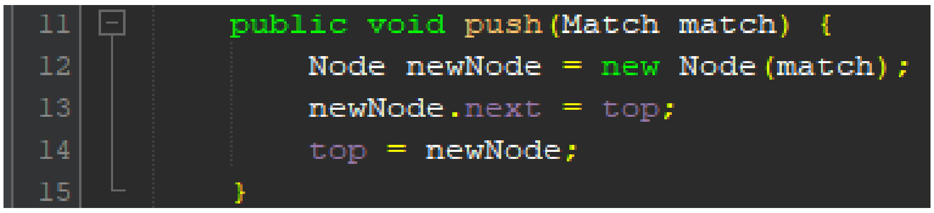


```java
    public MyStack() {
        top = null;
    }
```

Initializes an empty stack by setting top to null.

4. Methods

4.1 push()



```java
    public void push(Match match) {
        Node newNode = new Node(match);
        newNode.next = top;
        top = newNode;
    }
```

Purpose: Adds a new Match to the top of the stack.

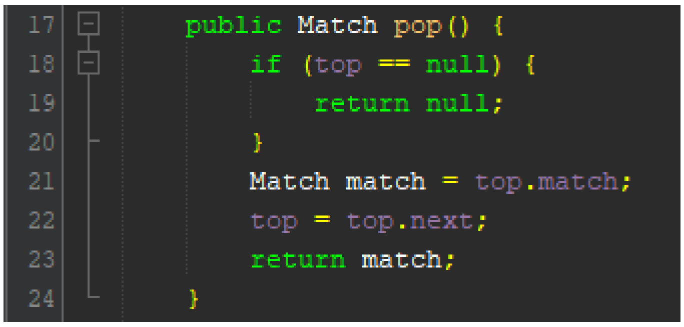
Steps:

Create a new Node that holds the provided Match object.

Set the next reference of the new node to the current top node. This links the new node to the rest of the stack.

Update top to point to the new node, making it the top element.

4.2 pop()



```java
    public Match pop() {
        if (top == null) {
            return null;
        }
        Match match = top.match;
        top = top.next;
        return match;
    }
```

Purpose: Removes and returns the top Match from the stack.

Steps:

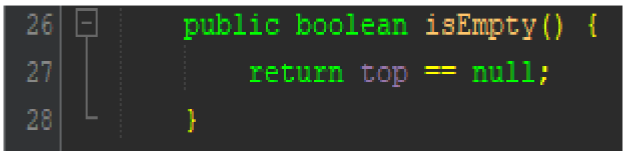
Check if the stack is empty (top == null). If so, return null.

Retrieve the Match object stored in the current top node.

Update top to point to the next node (top.Next). This effectively removes the current top node from the stack.

Return the retrieved Match.

4.3 isEmpty()

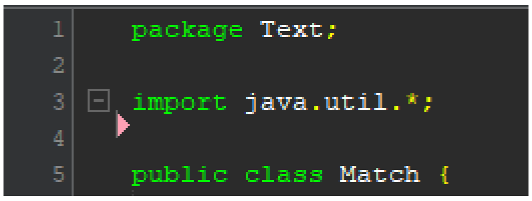


```java
    public boolean isEmpty() {
        return top == null;
    }
```

Purpose: Checks whether the stack is empty.
Returns:
true if top is null (i.e., no elements in the stack).
false otherwise.

MATCH CLASS

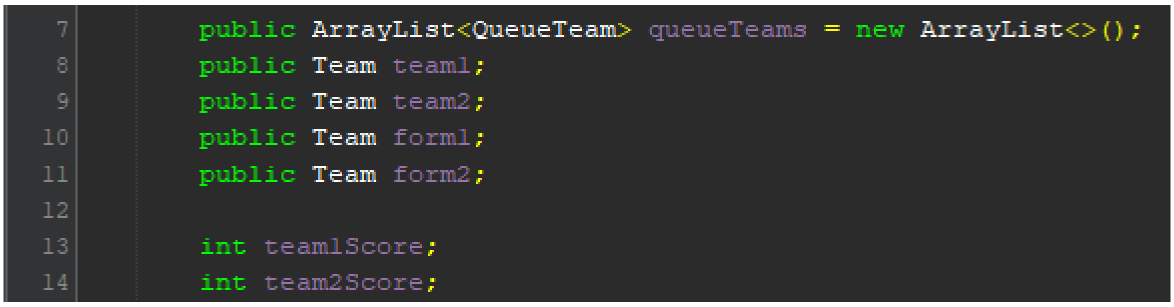
1. Class Definition



```java
package Text;

import java.util.*;

public class Match {
```

package Text: Indicates that this class is part of the Text package.
Imports:
java.util.*: For working with collections like ArrayList.

2. Fields



```java
    public ArrayList<QueueTeam> queueTeams = new ArrayList<>();
    public Team team1;
    public Team team2;
    public Team form1;
    public Team form2;

    int team1Score;
    int team2Score;
```

queueTeams: Holds a list of QueueTeam objects for managing fixtures.

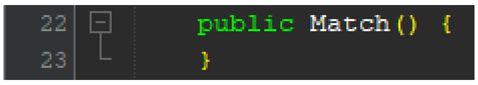
team1 & team2: Represent the two teams participating in a match.

form1 & form2: Placeholder for team forms (not actively used in the code).

team1Score & team2Score: Store the scores for team1 and team2 in a match

3. Constructor

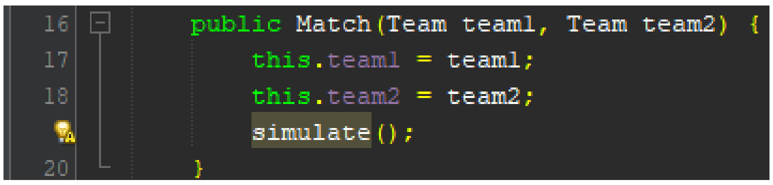
➔ **Default Constructor**



```java
    public Match() {
    }
```

A no-argument constructor for flexibility. Doesn't initialize any fields

➔ **Primary Constructor**

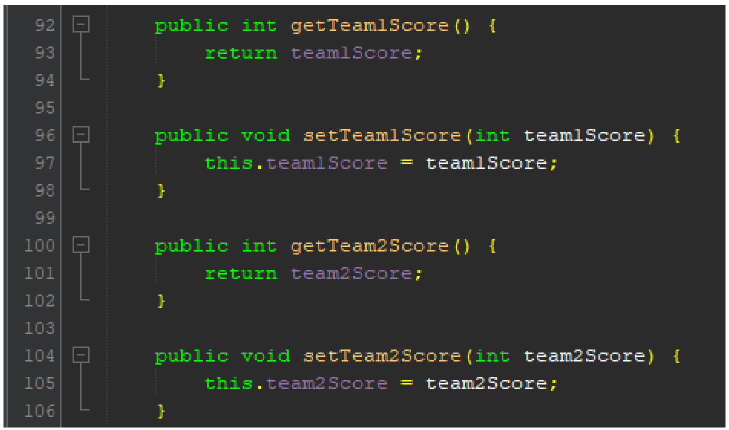


```java
    public Match(Team team1, Team team2) {
        this.team1 = team1;
        this.team2 = team2;
        simulate();
    }
```

Initializes a match between team1 and team2.

Calls the simulate() method to generate match results.

4. Getter and Setter Methods



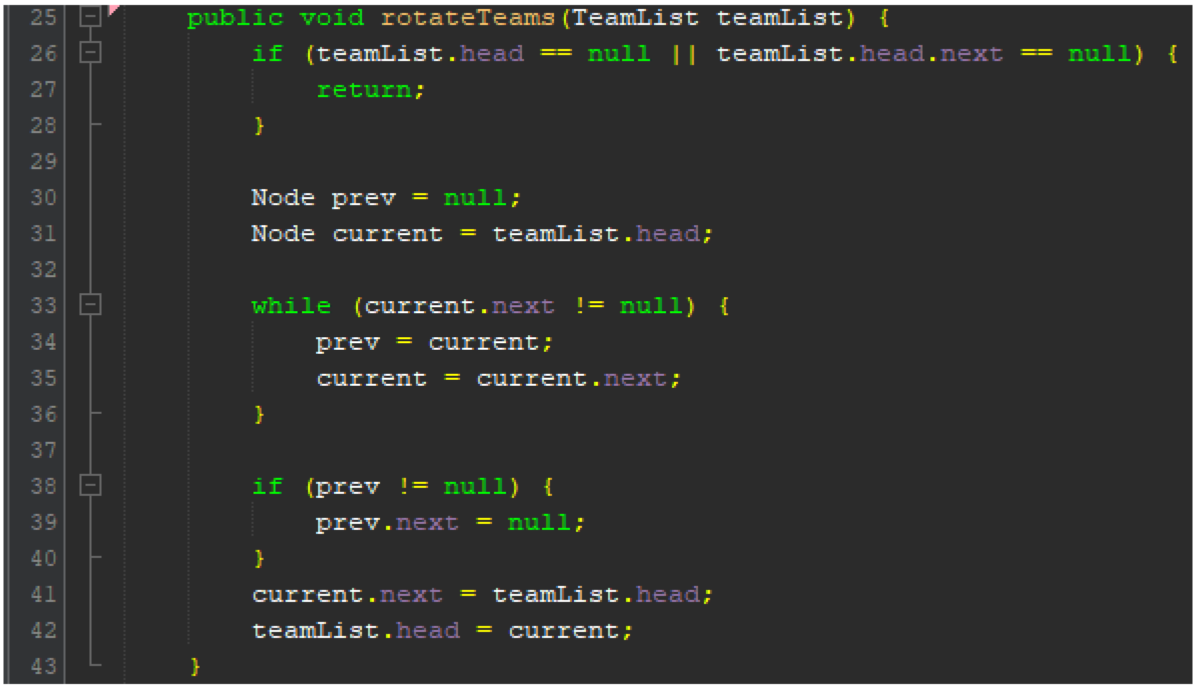
```java
    public int getTeam1Score() {
        return team1Score;
    }

    public void setTeam1Score(int team1Score) {
        this.team1Score = team1Score;
    }

    public int getTeam2Score() {
        return team2Score;
    }

    public void setTeam2Score(int team2Score) {
        this.team2Score = team2Score;
    }
```

Getters: Retrieve the scores for team1 and team2.

Setters: Manually update the scores for team1 and team2.

5. Methods

5.1 rotateTeams()



```java
    public void rotateTeams(TeamList teamList) {
        if (teamList.head == null || teamList.head.next == null) {
            return;
        }

        Node prev = null;
        Node current = teamList.head;

        while (current.next != null) {
            prev = current;
            current = current.next;
        }

        if (prev != null) {
            prev.next = null;
        }
        current.next = teamList.head;
        teamList.head = current;
    }
```

Purpose: Rotates the team list to generate a new round of fixtures.

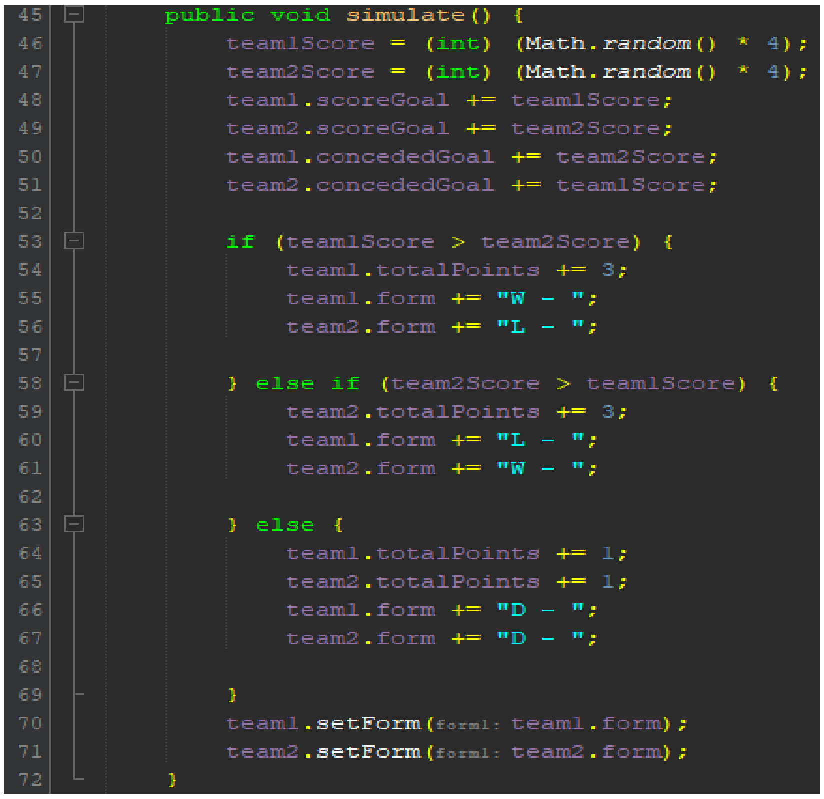
Steps:

Checks if the team list has at least two nodes (returns otherwise).

Iterates to the last node in the list.

Detaches the last node and makes it the new head of the list.

5.2 simulate()



```java
    public void simulate() {
        team1Score = (int) (Math.random() * 4);
        team2Score = (int) (Math.random() * 4);
        team1.scoreGoal += team1Score;
        team2.scoreGoal += team2Score;
        team1.concededGoal += team2Score;
        team2.concededGoal += team1Score;

        if (team1Score > team2Score) {
            team1.totalPoints += 3;
            team1.form += "W - ";
            team2.form += "L - ";

        } else if (team2Score > team1Score) {
            team2.totalPoints += 3;
            team1.form += "L - ";
            team2.form += "W - ";

        } else {
            team1.totalPoints += 1;
            team2.totalPoints += 1;
            team1.form += "D - ";
            team2.form += "D - ";

        }
        team1.setForm(form1: team1.form);
        team2.setForm(form1: team2.form);
    }
```

Purpose: Simulates a match and updates team stats.

Steps:

Randomly generates scores for both teams (0 to 3 goals).

Updates goals scored and conceded for both teams.

Updates total points and match form based on the result:

Win: +3 points.

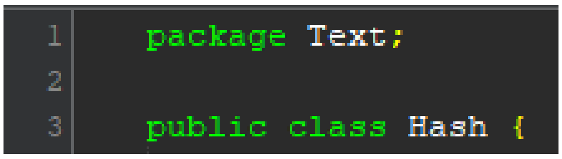
Draw: +1 point each.

Loss: No points.

Updates each team's form string.

HASH CLASS

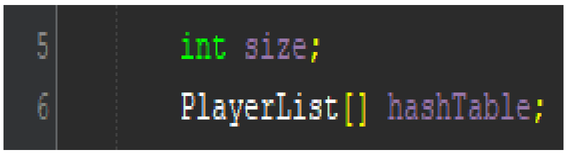
1. Class Definition



```java
package Text;

public class Hash {
```

package Text: The class is part of the Text package.
Purpose: Implements a simple hash table to store and retrieve PlayerList objects using string keys.

2. Fields

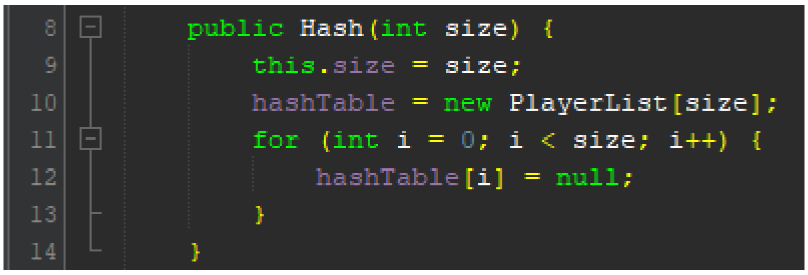


```java
    int size;
    PlayerList[] hashTable;
```

size: Represents the total size of the hash table (number of buckets).

hashTable: An array of PlayerList objects where each index acts as a bucket in the hash table.

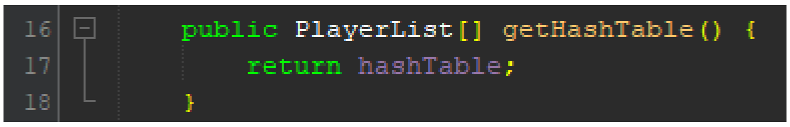
3. Constructor



```java
    public Hash(int size) {
        this.size = size;
        hashTable = new PlayerList[size];
        for (int i = 0; i < size; i++) {
            hashTable[i] = null;
        }
    }
```

Parameters:
size: The number of buckets in the hash table.
Initialization:
Creates an array of PlayerList objects with size elements.
Initializes each element to null.

4. Getter and Setter Methods

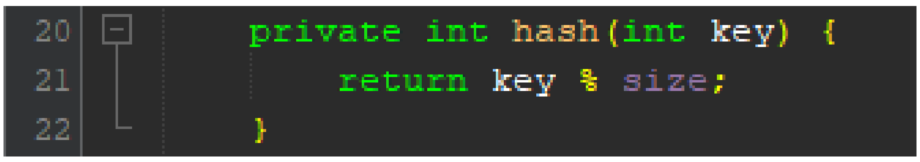


```java
    public PlayerList[] getHashTable() {
        return hashTable;
    }
```

Returns the entire hash table.

5. Methods

5.1 hash()



```java
    private int hash(int key) {
        return key % size;
    }
```

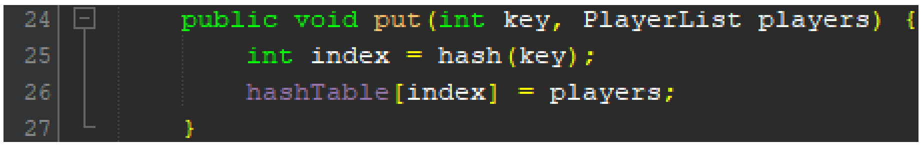
Input: Accepts an integer key.

Output: Returns the index in the hash table where the key-value pair will be stored.

Functionality:

Uses the modulus operator % to calculate the remainder when key is divided by size.

5.2 put()



```java
    public void put(int key, PlayerList players) {
        int index = hash(key);
        hashTable[index] = players;
    }
```

Parameters:

key: An integer used as the identifier.

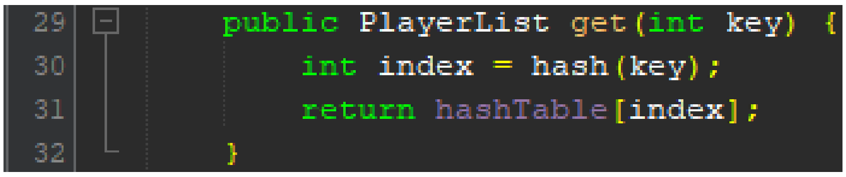
players: A PlayerList object to be stored in the hash table.

Functionality:

Calls the hash(key) function to compute the appropriate index.

Stores the players object at the computed index in the hashTable

5.3 get()



```java
    public PlayerList get(int key) {
        int index = hash(key);
        return hashTable[index];
    }
```

Parameter: Accepts an integer key.

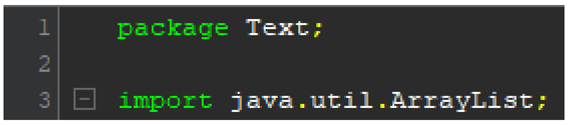
Functionality:

Computes the index for the given key using the hash function.

Returns the PlayerList object stored at that index in the hashTable.

HEAP CLASS

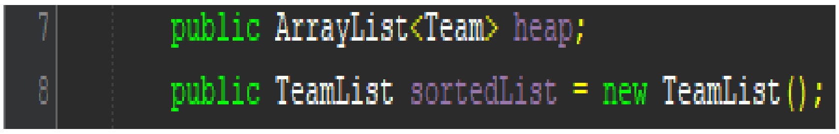
1. Class Definition



```java
package Text;

import java.util.ArrayList;
```

package Text: The class is part of the Text package.
Purpose: Implements a max heap to manage Team objects, based on their points and goal differences.

2. Fields

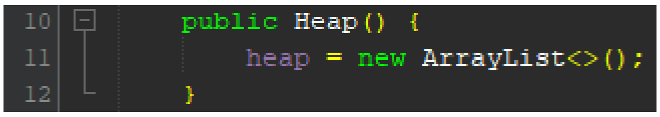


```java
    public ArrayList<Team> heap;
    public TeamList sortedList = new TeamList();
```

heap: An ArrayList storing Team objects in a heap structure.

sortedList: A TeamList to hold teams in sorted order (from highest to lowest) once extracted from the heap.

3. Constructor

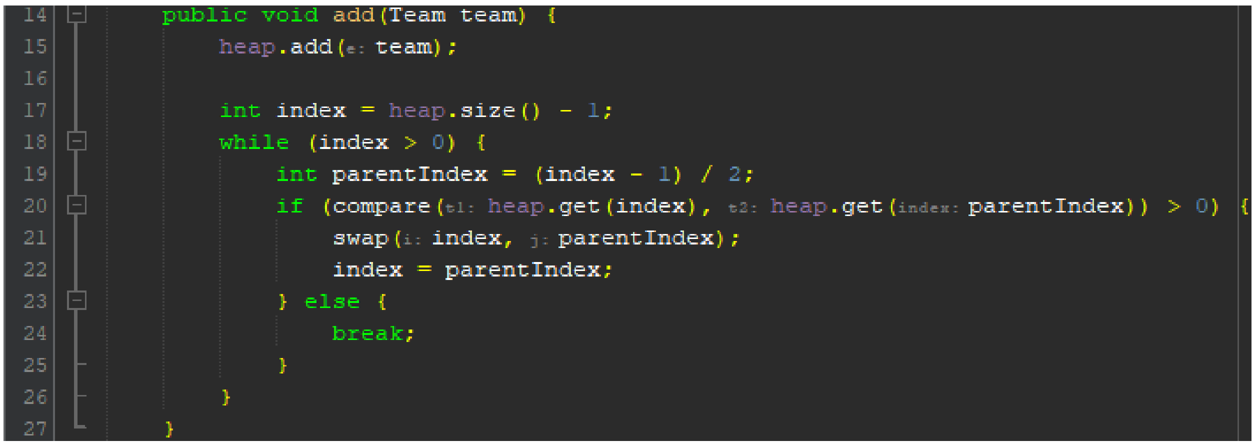


```java
public Heap() {
    heap = new ArrayList<>();
}
```

Initializes the heap as an empty ArrayList.

4. Methods

4.1 addPlayer()



```java
public void add(Team team) {
    heap.add(e: team);

    int index = heap.size() - 1;
    while (index > 0) {
        int parentIndex = (index - 1) / 2;
        if (compare(t1: heap.get(index), t2: heap.get(index: parentIndex)) > 0) {
            swap(i: index, j: parentIndex);
            index = parentIndex;
        } else {
            break;
        }
    }
}
```

Purpose: Adds a new Team to the heap and maintains the max-heap property.

Steps:

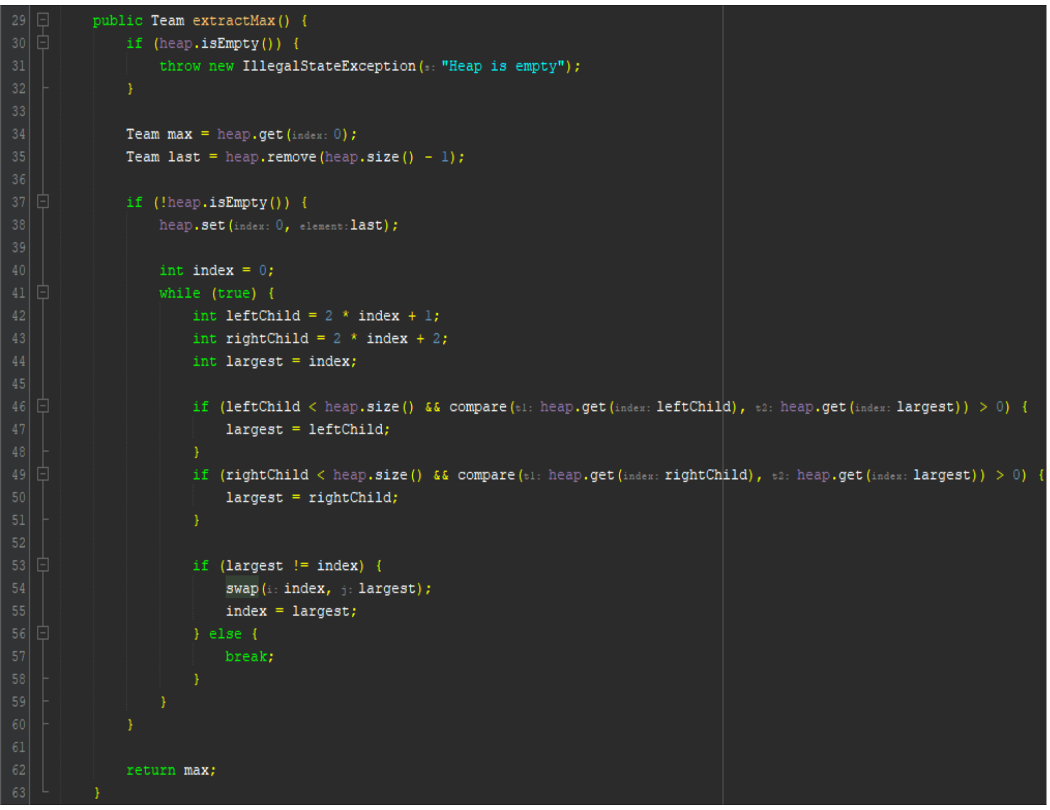
Add the team to the end of the heap.

Calculate its parent index: (index - 1) / 2.

If the new team is larger (based on compare), swap it with its parent.

Repeat until the max-heap property is restored or the root is reached.

4.2 extractMax()



```java
public Team extractMax() {
    if (heap.isEmpty()) {
        throw new IllegalStateException(s: "Heap is empty");
    }

    Team max = heap.get(index: 0);
    Team last = heap.remove(heap.size() - 1);

    if (!heap.isEmpty()) {
        heap.set(index: 0, element: last);

        int index = 0;
        while (true) {
            int leftChild = 2 * index + 1;
            int rightChild = 2 * index + 2;
            int largest = index;

            if (leftChild < heap.size() && compare(t1: heap.get(index: leftChild), t2: heap.get(index: largest)) > 0) {
                largest = leftChild;
            }
            if (rightChild < heap.size() && compare(t1: heap.get(index: rightChild), t2: heap.get(index: largest)) > 0) {
                largest = rightChild;
            }

            if (largest != index) {
                swap(i: index, j: largest);
                index = largest;
            } else {
                break;
            }
        }
    }

    return max;
}
```

Purpose: Removes and returns the largest Team (root) while maintaining the max-heap property.

Steps:

Store the root (heap.get(0)) as the largest value to return.

Replace the root with the last element.

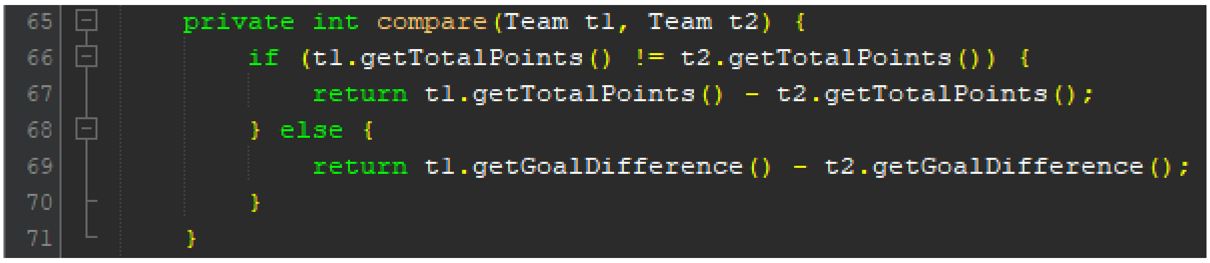
Remove the last element from the heap.

Restore the max-heap property by comparing the root with its children:

Swap with the largest child if necessary.

Repeat until the heap property is restored or no children exist.

4.3 compare()



```java
private int compare(Team t1, Team t2) {
    if (t1.getTotalPoints() != t2.getTotalPoints()) {
        return t1.getTotalPoints() - t2.getTotalPoints();
    } else {
        return t1.getGoalDifference() - t2.getGoalDifference();
    }
}
```

Purpose: Compares two teams based on:

Total points (getTotalPoints()).

Goal difference (getGoalDifference()) as a tiebreaker.

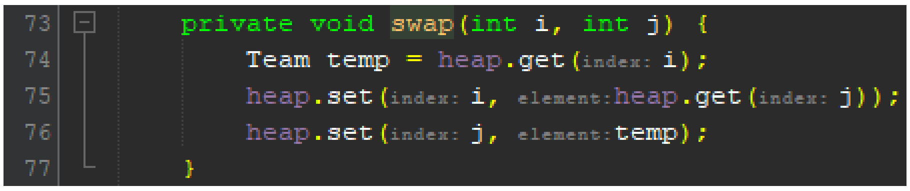
Returns:

A positive value if t1 is larger.

A negative value if t2 is larger.

0 if both are equal.

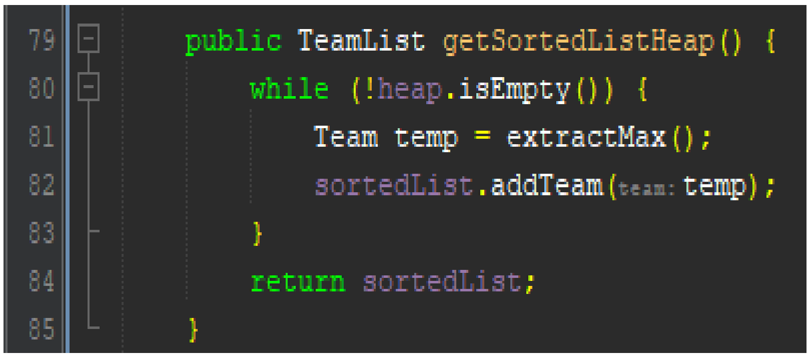
4.4 swap()



```java
private void swap(int i, int j) {
    Team temp = heap.get(index: i);
    heap.set(index: i, element: heap.get(index: j));
    heap.set(index: j, element: temp);
}
```

waps two elements in the heap at indices i and j

4.5 getSortedListHeap()



```java
public TeamList getSortedListHeap() {
    while (!heap.isEmpty()) {
        Team temp = extractMax();
        sortedList.addTeam(team: temp);
    }
    return sortedList;
}
```

Purpose: Extracts all elements from the heap in sorted order (highest to lowest) and adds them to sortedList.

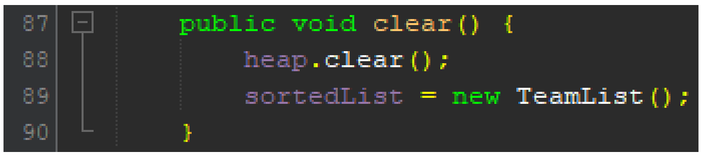
Steps:

Continuously calls extractMax to remove the largest element.

Adds the extracted Team to sortedList.

Returns the sortedList.

4.6 clear()

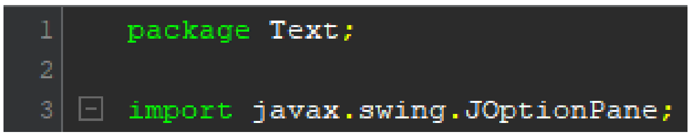


```java
public void clear() {
    heap.clear();
    sortedList = new TeamList();
}
```

Purpose: Clears the heap and resets sortedList

BST CLASS

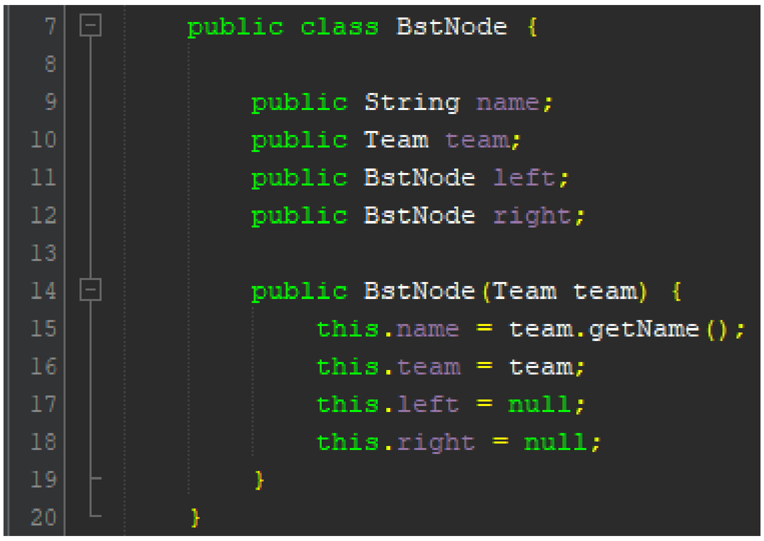
1. Class Definition



```java
package Text;

import javax.swing.JOptionPane;
```

Purpose: Implements a Binary Search Tree (BST) to store Team objects, organized by their names. Uses: The JOptionPane is used for GUI-based alerts.

2. Inner Class



```java
    public class BstNode {

        public String name;
        public Team team;
        public BstNode left;
        public BstNode right;

        public BstNode(Team team) {
            this.name = team.getName();
            this.team = team;
            this.left = null;
            this.right = null;
        }
    }
```

Purpose: Represents a node in the BST.

Fields:

name: Name of the team (used for comparisons).

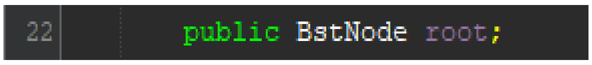
team: Reference to the Team object stored in this node.

left and right: Pointers to the left and right child nodes.

Constructor:

Initializes the node with a Team object and sets children to null.

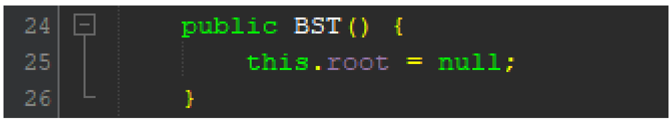
3. Fields



```java
    public BstNode root;
```

Purpose: Tracks the root of the BST.

4. Constructor

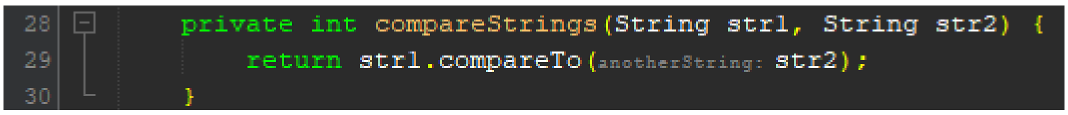


```java
    public BST() {
        this.root = null;
    }
```

Initializes an empty tree by setting the root to null

5. Methods

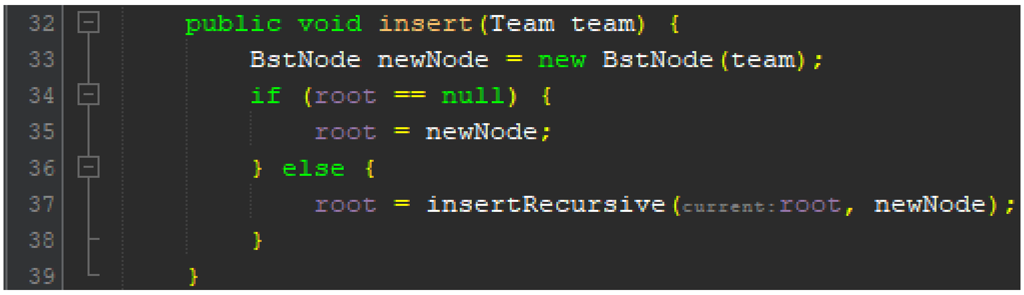
4.1 compareStrings()



```java
    private int compareStrings(String str1, String str2) {
        return str1.compareTo(anotherString: str2);
    }
```

Purpose: Compares two strings lexicographically.
Returns:
Negative if str1 < str2.
Zero if str1 == str2.
Positive if str1 > str2.

4.2 insert()



```java
public void insert(Team team) {
    BstNode newNode = new BstNode(team);
    if (root == null) {
        root = newNode;
    } else {
        root = insertRecursive(current:root, newNode);
    }
}
```

Purpose: Inserts a Team object into the BST.

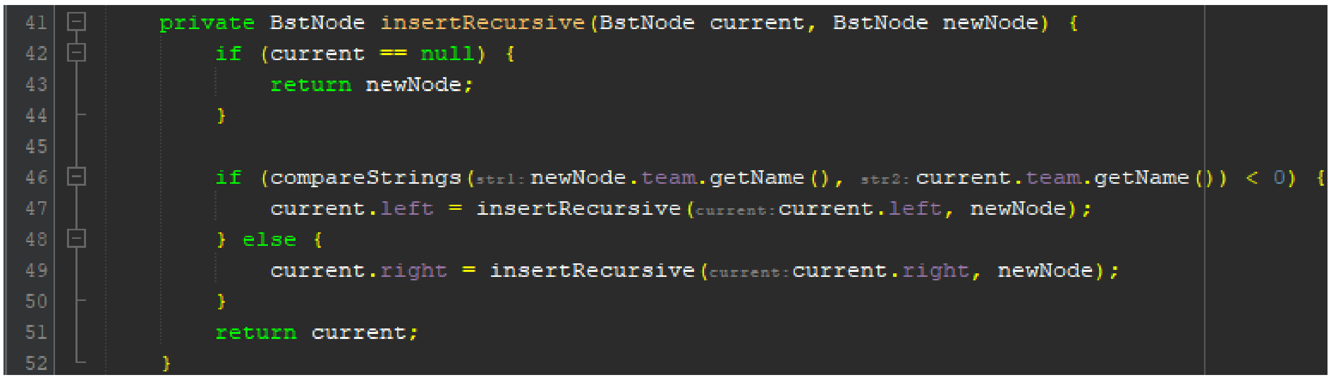
Steps:

Create a new BstNode for the team.

If the tree is empty (root == null), set the new node as the root.

Otherwise, call insertRecursive to find the correct position for the new node.

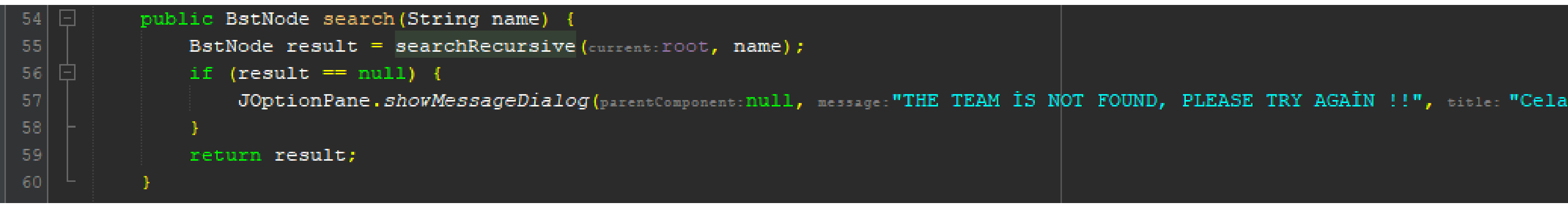
4.3 insertRecursive()



```java
private BstNode insertRecursive(BstNode current, BstNode newNode) {
    if (current == null) {
        return newNode;
    }

    if (compareStrings(str1: newNode.team.getName(), str2: current.team.getName()) < 0) {
        current.left = insertRecursive(current:current.left, newNode);
    } else {
        current.right = insertRecursive(current:current.right, newNode);
    }
    return current;
}
```

Purpose: Recursively inserts a node in the correct position.
Steps:
If current is null, return the new node (base case).
Compare newNode's name with current's name.
If less, recursively insert into the left subtree.
If greater or equal, recursively insert into the right subtree.
Return the updated current node to maintain links.

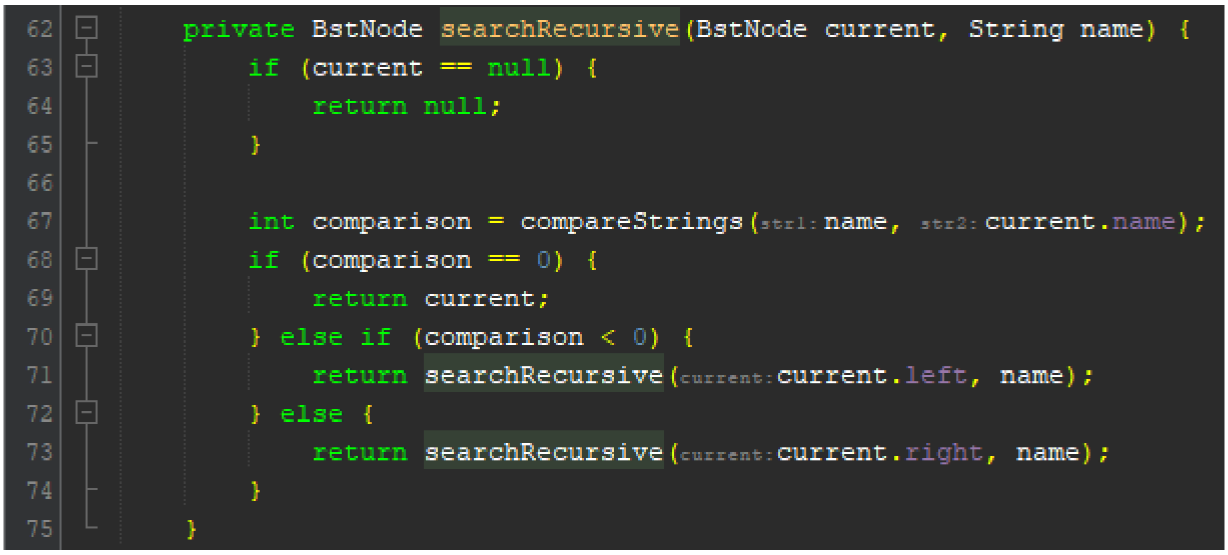
4.4 search()



```java
public BstNode search(String name) {
    BstNode result = searchRecursive(current:root, name);
    if (result == null) {
        JOptionPane.showMessageDialog(parentComponent:null, message:"THE TEAM İS NOT FOUND, PLEASE TRY AGAİN !!", title: "Cela
    }
    return result;
}
```

Purpose: Searches for a team by name in the BST.
Steps:
Calls searchRecursive to perform the search.
If the result is null, shows a GUI message that the team was not found.
Returns the found node or null.

4.5 searchRecursive()



```java
    private BstNode searchRecursive(BstNode current, String name) {
        if (current == null) {
            return null;
        }

        int comparison = compareStrings( str1: name,  str2: current.name);
        if (comparison == 0) {
            return current;
        } else if (comparison < 0) {
            return searchRecursive( current: current.left, name);
        } else {
            return searchRecursive( current: current.right, name);
        }
    }
```

Purpose: Recursively searches for a node with the given name.

Steps:

If current is null, return null (base case).

Compare the target name with current's name:

If equal, return the current node.

If less, search in the left subtree.

If greater, search in the right subtree.

main CLASS

The main class initializes a Swing-based GUI with two panels:

menuPanel: Manages menu-related components using CardLayout.

mainPanel: Switches between leaguePanel and teamPanel using CardLayout.

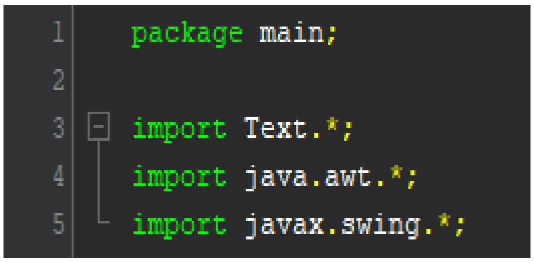
Initially:

menuPanel displays the menu component.

mainPanel displays the leaguePanel component.

The backLeague button is hidden.

1. Package and Imports



```java
package main;

import Text.*;
import java.awt.*;
import javax.swing.*;
```

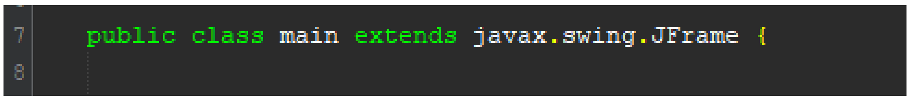
package main; Indicates that this class belongs to the main package.

import Text.*; Imports all classes from the Text package.

import java.awt.*; Imports AWT classes such as CardLayout and Color.

import javax.swing.*; Imports Swing components like JFrame, JPanel, etc.

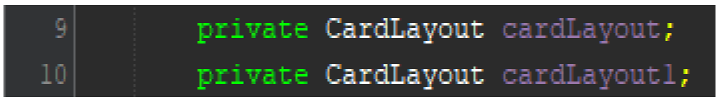
2. Class Declaration



```java
public class main extends javax.swing.JFrame {

```

The main class extends JFrame, meaning it represents a window in a Swing-based GUI application.

3. Instance Variables

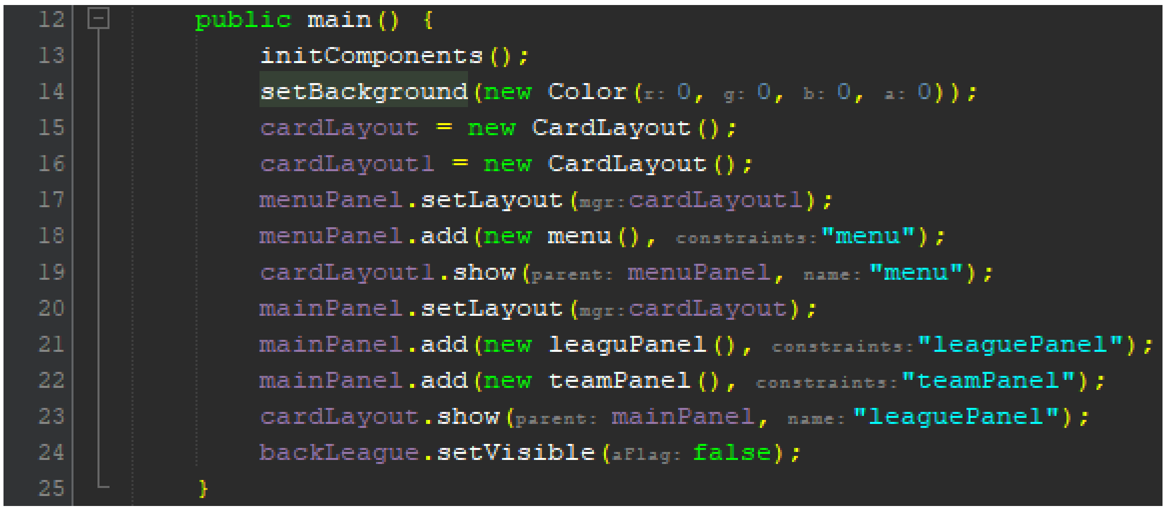


```java
    private CardLayout cardLayout;
    private CardLayout cardLayout1;
```

cardLayout: Manages the switching of panels within mainPanel.

cardLayout1: Manages the switching of panels within menuPanel.

4. Constructor



```java
    public main() {
        initComponents();
        setBackground(new Color(r: 0, g: 0, b: 0, a: 0));
        cardLayout = new CardLayout();
        cardLayout1 = new CardLayout();
        menuPanel.setLayout(mgr:cardLayout1);
        menuPanel.add(new menu(), constraints:"menu");
        cardLayout1.show(parent: menuPanel, name: "menu");
        mainPanel.setLayout(mgr:cardLayout);
        mainPanel.add(new leaguPanel(), constraints:"leaguePanel");
        mainPanel.add(new teamPanel(), constraints:"teamPanel");
        cardLayout.show(parent: mainPanel, name: "leaguePanel");
        backLeague.setVisible(aFlag: false);
    }
```

The constructor initializes the GUI and sets up the layout and components.

initComponents():

Likely auto-generated by a GUI builder (e.g., NetBeans).

Sets up the basic JFrame structure and components such as menuPanel, mainPanel, and backLeague.

setBackground(new Color(0, 0, 0, 0)):

Sets the window's background color to a transparent black (RGBA values: red, green, blue, alpha).

cardLayout:

Used for switching between panels in the mainPanel.

cardLayout1:

Used for switching between panels in the menuPanel.

menuPanel.setLayout(cardLayout1):

Sets the layout of menuPanel to CardLayout, allowing dynamic switching of components.

menuPanel.add(new menu(), "menu"):

Adds a menu object (likely a custom JPanel or component) to the menuPanel with the identifier "menu".

cardLayout1.show(menuPanel, "menu"):

Displays the menu component in menuPanel.

mainPanel.setLayout(cardLayout):

Sets the layout of mainPanel to CardLayout.

mainPanel.add(new leaguPanel(), "leaguePanel"):

Adds a leaguPanel (likely a custom JPanel or component for league-related functionality) to mainPanel with the identifier "leaguePanel".

mainPanel.add(new teamPanel(), "teamPanel"):

Adds a teamPanel (likely a custom JPanel or component for team-related functionality) to mainPanel with the identifier "teamPanel".

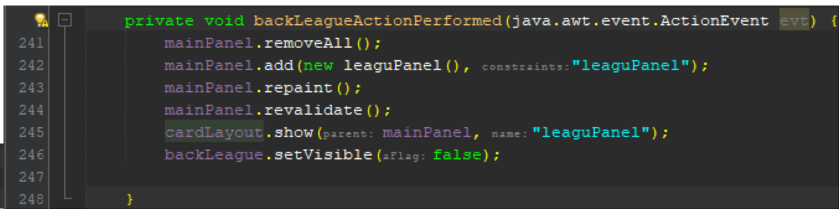
cardLayout.show(mainPanel, "leaguePanel"):

Displays the leaguePanel component in mainPanel.

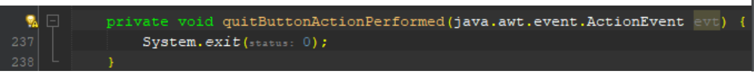
backLeague.setVisible(false):

Hides the backLeague button or component. This button may be intended for navigation and will likely become visible based on user interactions.

mainPanel :



```
private void backLeagueActionPerformed(java.awt.event.ActionEvent evt) {
    mainPanel.removeAll();
    mainPanel.add(new leaguPanel(), constraints:"leaguPanel");
    mainPanel.repaint();
    mainPanel.revalidate();
    cardLayout.show(parent: mainPanel, name: "leaguPanel");
    backLeague.setVisible(aFlag: false);

}
```



```
private void quitButtonActionPerformed(java.awt.event.ActionEvent evt) {
    System.exit(status: 0);
}
```

quitButton: User exit the system.

backLeague: When the "Back to League" button is clicked

It removes all existing content from mainPanel.

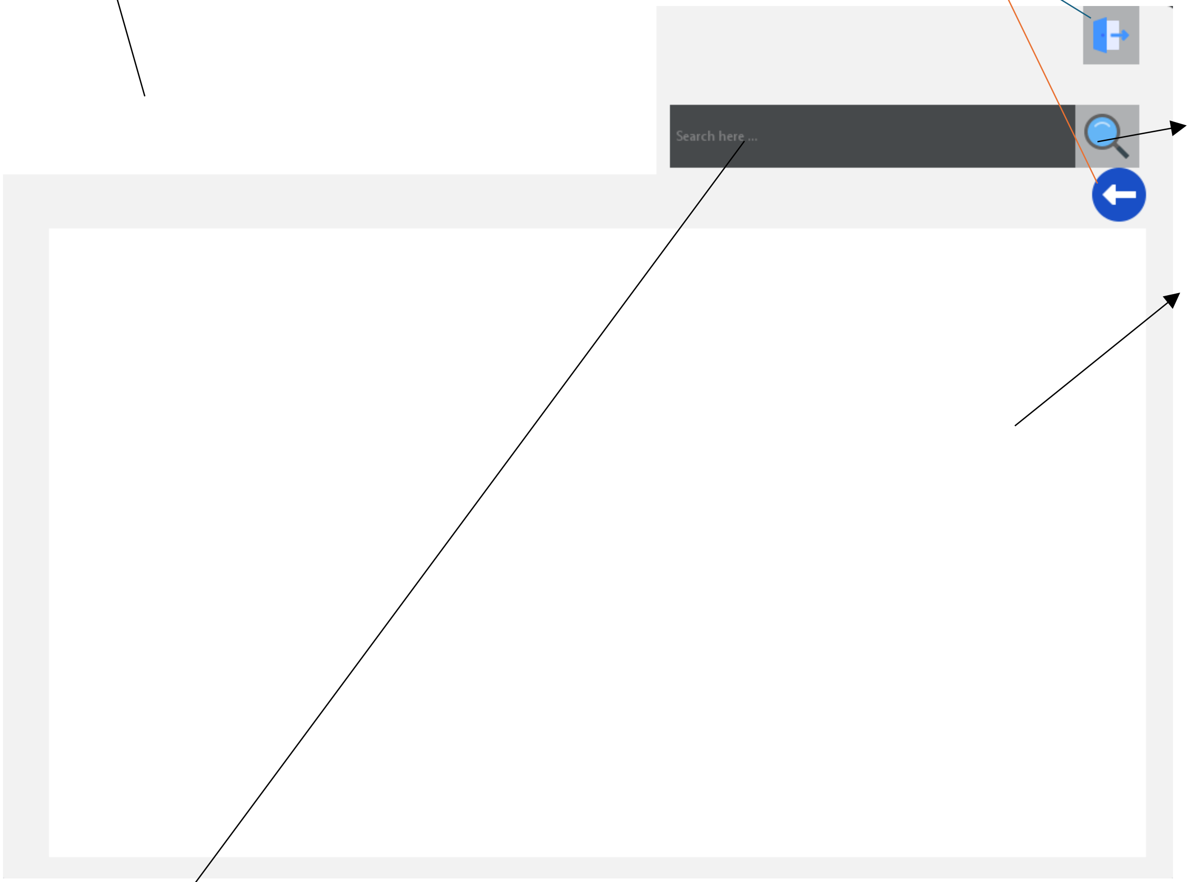
It adds a new leaguPanel to the mainPanel and identifies it with the string "leaguPanel".

The layout is updated and the new leaguPanel is displayed.

The "Back to League" button is hidden, likely to prevent the user from returning to this view again in the current context.

This functionality is typically used to navigate back to the main league screen after viewing or interacting with a different panel (e.g., the teamPanel).

menuPanel: We used it as cardLayout1

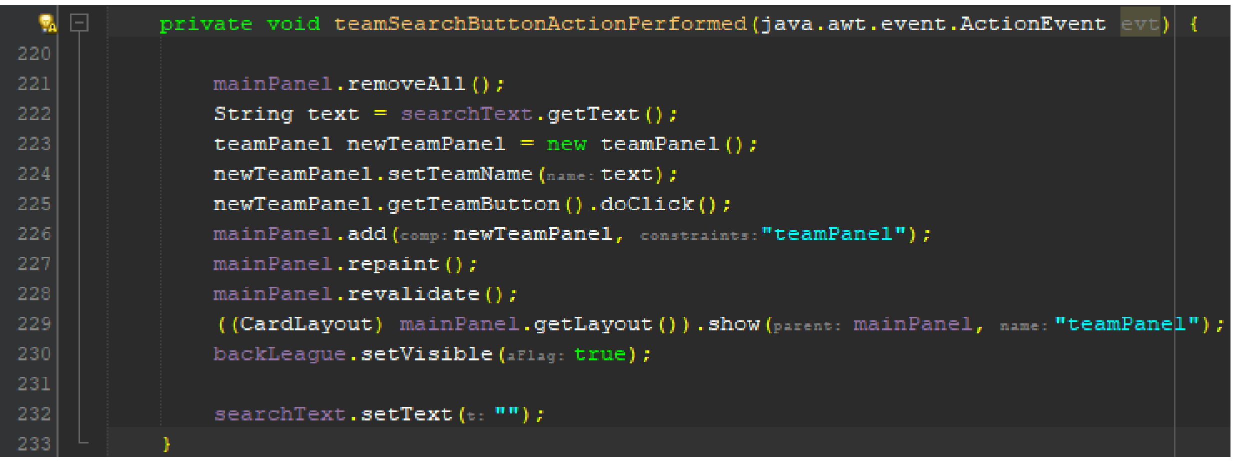


teamSearchButton:

mainPanel: We used it as cardLayout. teamPanel and leaguepanel will come here

searchText: Enter the team whose player list you want to see.

teamSearchButton:



```java
    private void teamSearchButtonActionPerformed(java.awt.event.ActionEvent evt) {

        mainPanel.removeAll();
        String text = searchText.getText();
        teamPanel newTeamPanel = new teamPanel();
        newTeamPanel.setTeamName(name: text);
        newTeamPanel.getTeamButton().doClick();
        mainPanel.add(comp: newTeamPanel, constraints: "teamPanel");
        mainPanel.repaint();
        mainPanel.revalidate();
        ((CardLayout) mainPanel.getLayout()).show(parent: mainPanel, name: "teamPanel");
        backLeague.setVisible(aFlag: true);

        searchText.setText(t: "");
    }
```

This method is triggered when the user clicks the team search button. Here's what it does:

Clears any existing content in mainPanel.

Retrieves the team's name or search keyword from searchText.

Creates a new teamPanel and sets the team's name based on the search input.

Simulates a click on a button within the teamPanel.

Adds the new teamPanel to mainPanel and displays it.

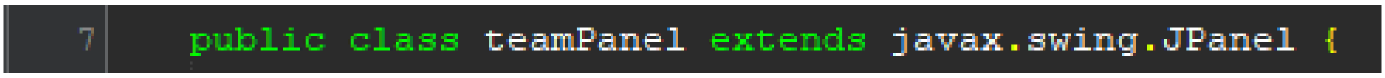
Makes the backLeague button visible (for navigation).

Clear the search text field to prepare for the next search.

teamPanel CLASS

This class is designed to handle the creation and management of teams and their players. It uses a combination of Player, PlayerList, BST, and Hash data structures to organize and store player and team data. Additionally, it prepares the necessary data structures for the teamPanel GUI component, making it easier to manipulate and display team-related data.

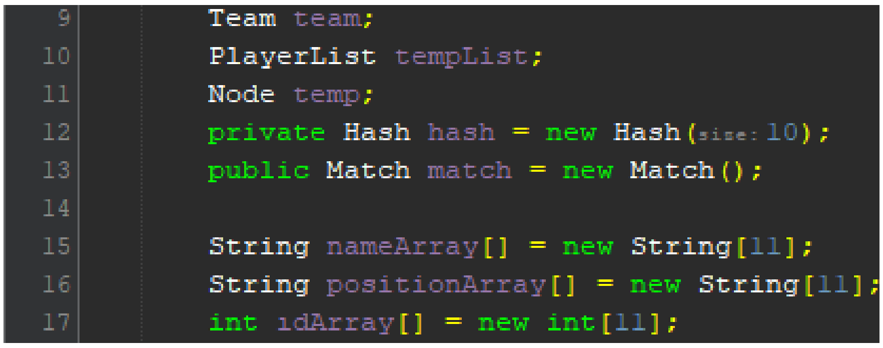
1. Class Declaration



```java
public class teamPanel extends javax.swing.JPanel {
```

The class teamPanel extends JPanel, meaning it is a GUI component that can be added to a JFrame or other containers in the application

2. Instance Variables



```java
Team team;
PlayerList tempList;
Node temp;
private Hash hash = new Hash(size: 10);
public Match match = new Match();

String nameArray[] = new String[11];
String positionArray[] = new String[11];
int idArray[] = new int[11];
```

The class defines various instance variables for handling the team, players, and other relevant data structures:

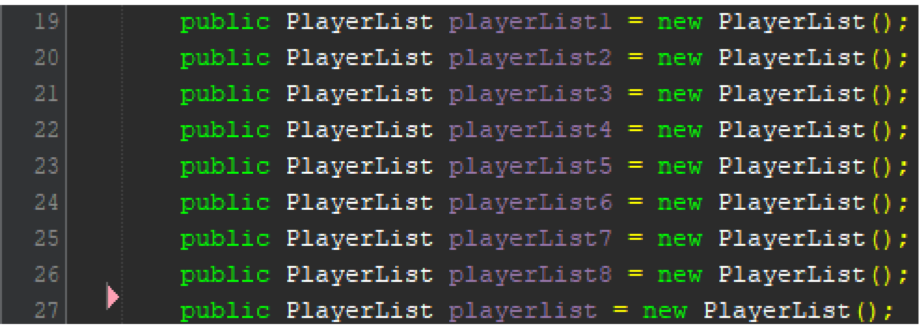
Team team: Represents a single team.

PlayerList and Node temp: These store the list of players for the team and some nodes for data management.

Hash hash = new Hash(10): Creates a hash table to store the teams' player lists by team name.

Other arrays (**nameArray, positionArray, etc**.) are used to store details about players like their names, positions, goals, and IDs.

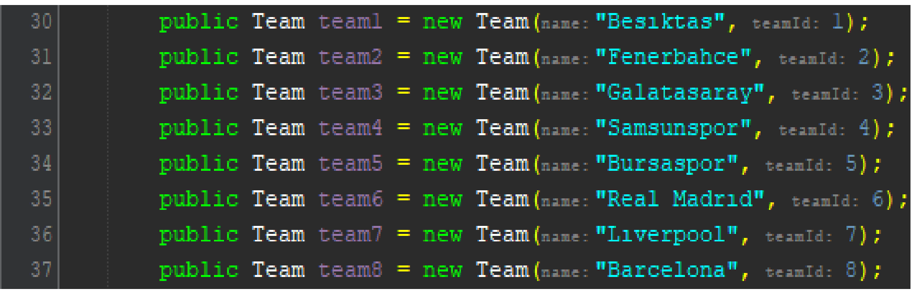
List of Players:



```java
public PlayerList playerList1 = new PlayerList();
public PlayerList playerList2 = new PlayerList();
public PlayerList playerList3 = new PlayerList();
public PlayerList playerList4 = new PlayerList();
public PlayerList playerList5 = new PlayerList();
public PlayerList playerList6 = new PlayerList();
public PlayerList playerList7 = new PlayerList();
public PlayerList playerList8 = new PlayerList();
public PlayerList playerlist = new PlayerList();
```

The PlayerList instances like playerList1, playerList2, etc., are used to store players for different teams. These lists are populated later in the code.

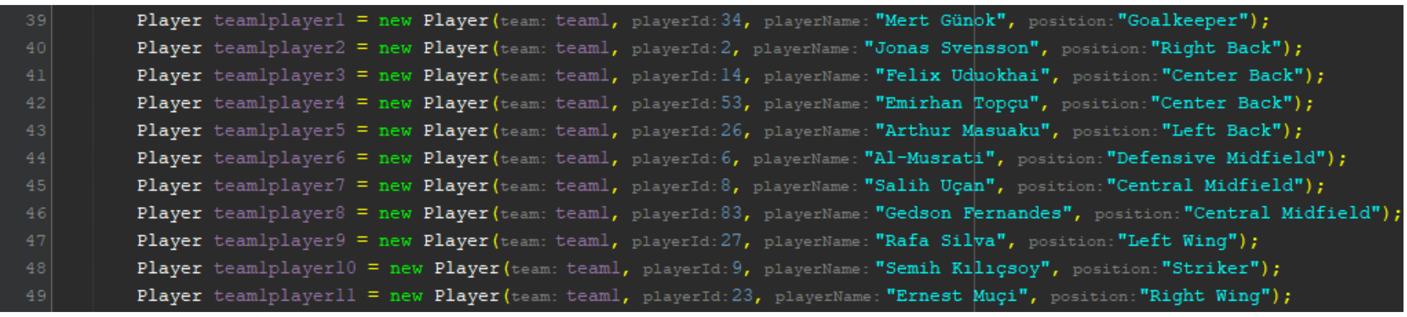
Team Creation:



```java
public Team team1 = new Team(name: "Besiktas", teamId: 1);
public Team team2 = new Team(name: "Fenerbahce", teamId: 2);
public Team team3 = new Team(name: "Galatasaray", teamId: 3);
public Team team4 = new Team(name: "Samsunspor", teamId: 4);
public Team team5 = new Team(name: "Bursaspor", teamId: 5);
public Team team6 = new Team(name: "Real Madrid", teamId: 6);
public Team team7 = new Team(name: "Liverpool", teamId: 7);
public Team team8 = new Team(name: "Barcelona", teamId: 8);
```

The teams are created as instances of the Team class.

Player Creation:

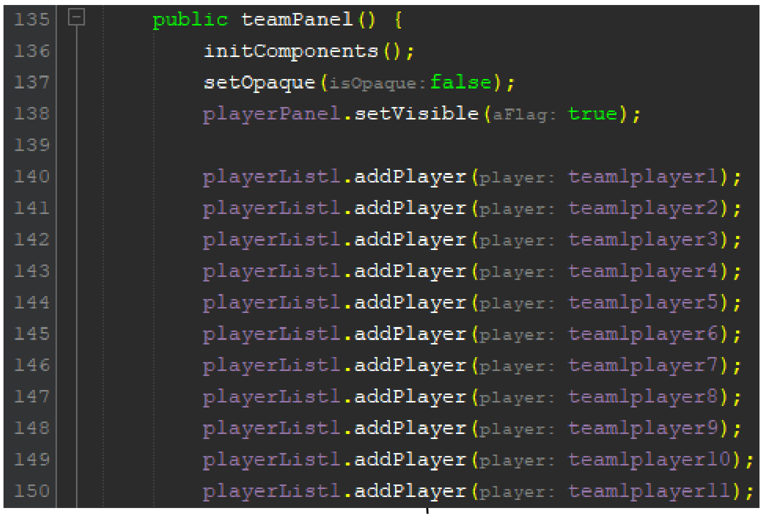


```java
Player team1player1 = new Player(team: team1, playerId:34, playerName:"Mert Günok", position:"Goalkeeper");
Player team1player2 = new Player(team: team1, playerId:2, playerName:"Jonas Svensson", position:"Right Back");
Player team1player3 = new Player(team: team1, playerId:14, playerName:"Felix Uduokhai", position:"Center Back");
Player team1player4 = new Player(team: team1, playerId:53, playerName:"Emirhan Topçu", position:"Center Back");
Player team1player5 = new Player(team: team1, playerId:26, playerName:"Arthur Masuaku", position:"Left Back");
Player team1player6 = new Player(team: team1, playerId:6, playerName:"Al-Musrati", position:"Defensive Midfield");
Player team1player7 = new Player(team: team1, playerId:8, playerName:"Salih Uçan", position:"Central Midfield");
Player team1player8 = new Player(team: team1, playerId:83, playerName:"Gedson Fernandes", position:"Central Midfield");
Player team1player9 = new Player(team: team1, playerId:27, playerName:"Rafa Silva", position:"Left Wing");
Player team1player10 = new Player(team: team1, playerId:9, playerName:"Semih Kılıçsoy", position:"Striker");
Player team1player11 = new Player(team: team1, playerId:23, playerName:"Ernest Muçi", position:"Right Wing");
```

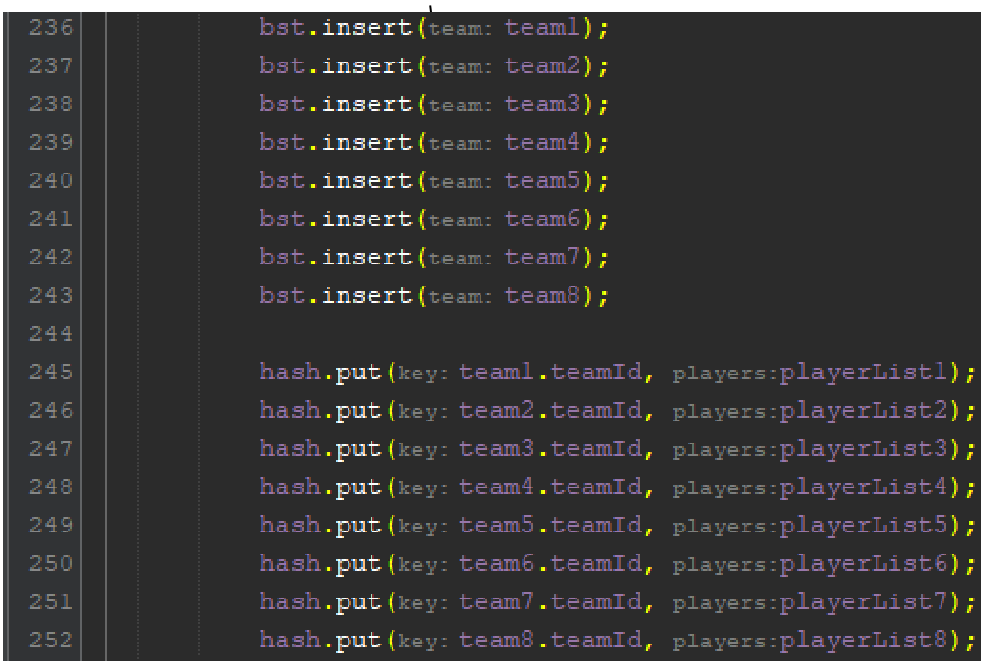
For each team, 11 players are created as instances of the Player class:

Each player is associated with the team, assigned a number, a name, and a position

3. Constructor



```
public teamPanel() {
    initComponents();
    setOpaque(isOpaque:false);
    playerPanel.setVisible(aFlag: true);

    playerList1.addPlayer(player: team1player1);
    playerList1.addPlayer(player: team1player2);
    playerList1.addPlayer(player: team1player3);
    playerList1.addPlayer(player: team1player4);
    playerList1.addPlayer(player: team1player5);
    playerList1.addPlayer(player: team1player6);
    playerList1.addPlayer(player: team1player7);
    playerList1.addPlayer(player: team1player8);
    playerList1.addPlayer(player: team1player9);
    playerList1.addPlayer(player: team1player10);
    playerList1.addPlayer(player: team1player11);
```



```
        bst.insert(team: team1);
        bst.insert(team: team2);
        bst.insert(team: team3);
        bst.insert(team: team4);
        bst.insert(team: team5);
        bst.insert(team: team6);
        bst.insert(team: team7);
        bst.insert(team: team8);

        hash.put(key: team1.teamId, players:playerList1);
        hash.put(key: team2.teamId, players:playerList2);
        hash.put(key: team3.teamId, players:playerList3);
        hash.put(key: team4.teamId, players:playerList4);
        hash.put(key: team5.teamId, players:playerList5);
        hash.put(key: team6.teamId, players:playerList6);
        hash.put(key: team7.teamId, players:playerList7);
        hash.put(key: team8.teamId, players:playerList8);
```

teamPanel Constructor:

The constructor method (teamPanel()) is responsible for initializing all the components:

It calls initComponents() to set up GUI components (this is typically generated by the IDE).

setOpaque(false) makes the panel transparent.

playerPanel.setVisible(true) ensures the player panel is visible.

Adding Players to Player Lists: Players are added to their respective PlayerList objects.

BST (Binary Search Tree) Insertion: The teams are inserted into a binary search tree (bst):

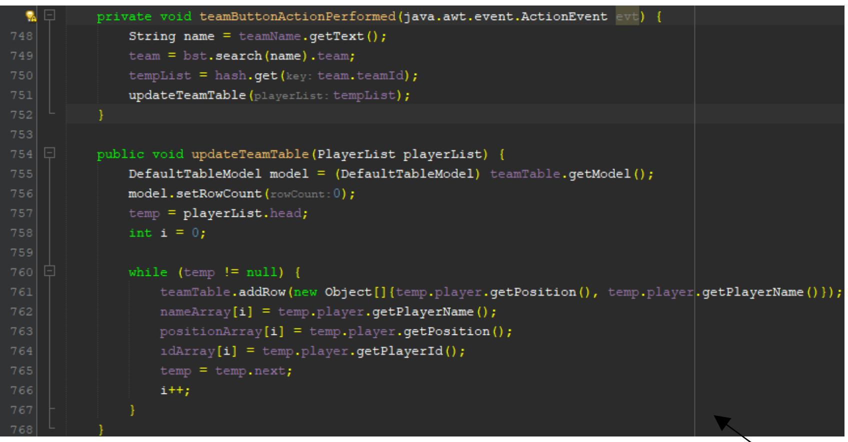
The BST is used to organize the teams in a search-friendly manner, allowing for efficient access and retrieval.

Hash Table Insertion: The player lists are associated with each team's name in a hash table:

This allows you to retrieve the list of players for any given team by using the team's name.

teamPanel:

teamButton:



```
    private void teamButtonActionPerformed(java.awt.event.ActionEvent evt) {
        String name = teamName.getText();
        team = bst.search(name).team;
        tempList = hash.get(key: team.teamId);
        updateTeamTable(playerList: tempList);
    }

    public void updateTeamTable(PlayerList playerList) {
        DefaultTableModel model = (DefaultTableModel) teamTable.getModel();
        model.setRowCount(rowCount: 0);
        temp = playerList.head;
        int i = 0;

        while (temp != null) {
            teamTable.addRow(new Object[]{temp.player.getPosition(), temp.player.getPlayerName()});
            nameArray[i] = temp.player.getPlayerName();
            positionArray[i] = temp.player.getPosition();
            ıdArray[i] = temp.player.getPlayerId();
            temp = temp.next;
            i++;
        }
    }
```

When the button is clicked: The method retrieves the team’s name from the user input. It searches the BST for the corresponding Team object. If the team is found, it fetches the associated player list from the hash table. Finally, it updates the GUI to display the players of the selected team.

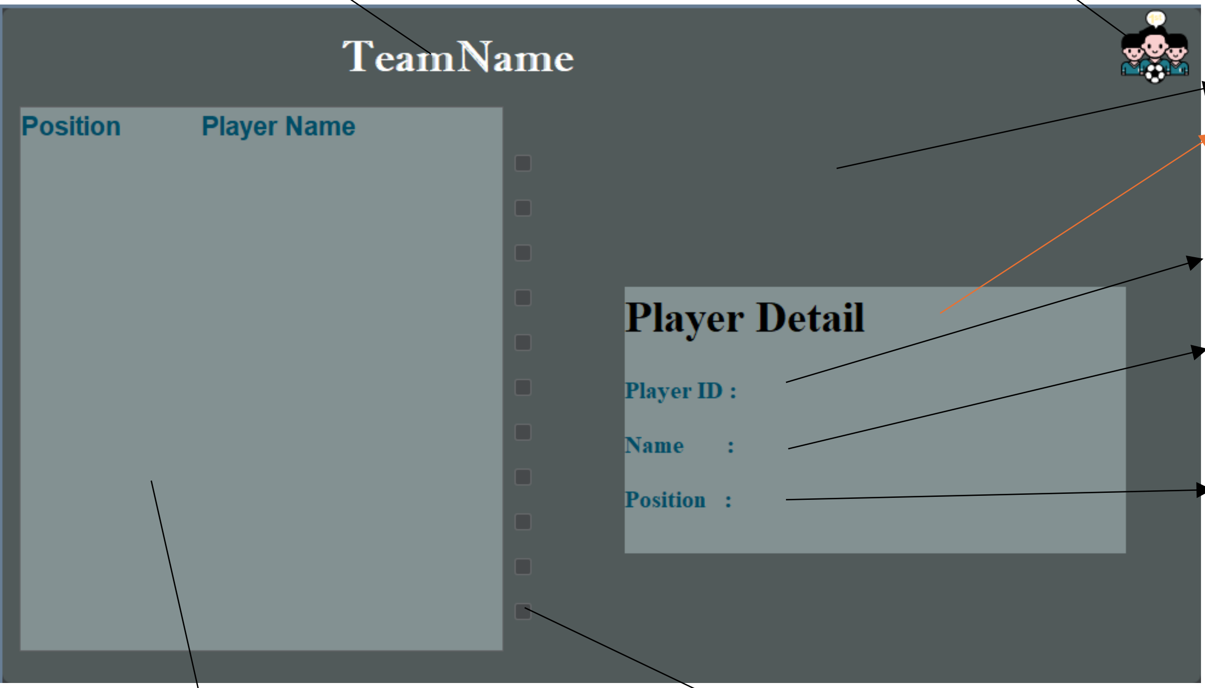
The updateTeamTable method:
Clears the Table: Removes all previous rows from the table.
Traverses the Linked List: Iterates through the PlayerList.
Adds Rows: Inserts each player's position and name into the table.
Updates Arrays: Fills arrays (nameArray, positionArray, etc.) with player data.

Error Handling: Can handle empty list cases if implemented.

Correct Method Usage: Uses the table model (model.addRow) for proper data insertion.

Summary: Transfers player data from the linked list to the table and associated arrays.

teamName: The team’s name entered in the search text is written here



teampanel:

playerPanel:

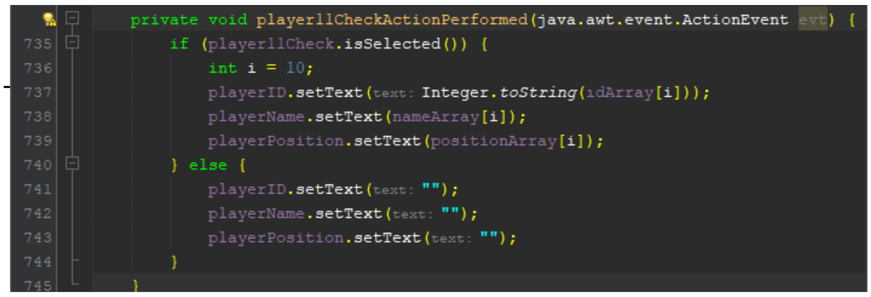
playerId: player id is written

playerName: player name is written

playerPosition: player position is written

teamTable: The positions and names of the players are written in this table.

player11Check:



```
    private void player11CheckActionPerformed(java.awt.event.ActionEvent evt) {
        if (player11Check.isSelected()) {
            int i = 10;
            playerID.setText(text: Integer.toString(ıdArray[i]));
            playerName.setText(nameArray[i]);
            playerPosition.setText(positionArray[i]);
        } else {
            playerID.setText(text: "");
            playerName.setText(text: "");
            playerPosition.setText(text: "");
        }
    }
```

Checkbox Checked: If the checkbox is selected (player11Check.isSelected()): The player's data for index 10 (11th player, as arrays are zero-indexed) is retrieved from the ıdArray, nameArray, and positionArray. The corresponding fields (playerID, playerName, and playerPosition) are updated to display the selected player's information.

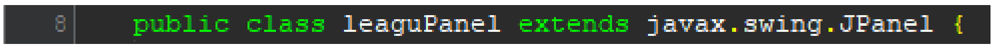
Checkbox Unchecked: If the checkbox is deselected:

The fields are cleared by setting their text to an empty string ("").

leaguePanel CLASS

This code simulates a league management system by creating a panel in a Java Swing application. Here's a step-by-step explanation of how the code works:

1. Class Definition

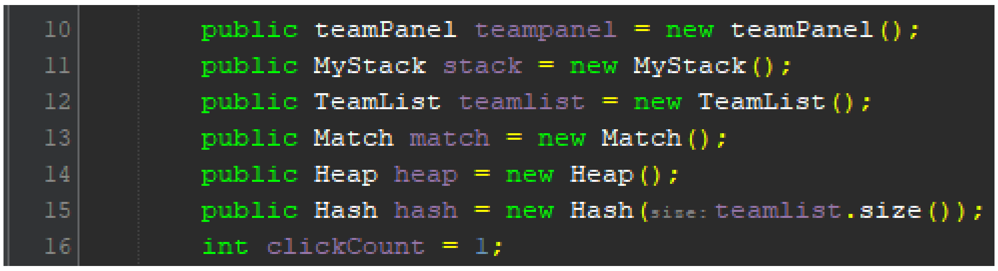


```java
  public class leaguPanel extends javax.swing.JPanel {
```

The leaguPanel class extends the JPanel class and is used as a Swing UI component.

This class represents a league panel.

2. Instance Variables



```java
    public teamPanel teampanel = new teamPanel();
    public MyStack stack = new MyStack();
    public TeamList teamlist = new TeamList();
    public Match match = new Match();
    public Heap heap = new Heap();
    public Hash hash = new Hash(size: teamlist.size());
    int clickCount = 1;
```

<u>teamPanel:</u> Manages teams and players.

<u>stack</u>: A custom data structure, possibly for storing past operations.

<u>teamlist:</u> A class where teams are stored in a linked list structure.

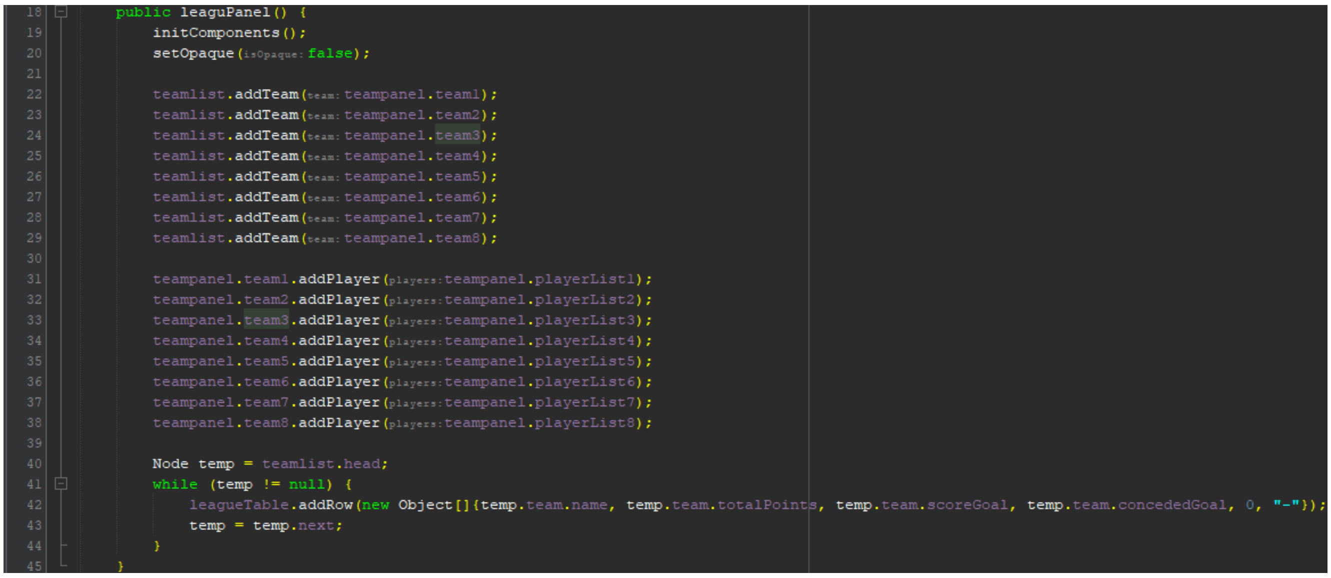
<u>match:</u> Represents match data.

<u>heap:</u> Likely used for sorting matches or scores.

<u>hash:</u> A hash table for quickly searching teams or players.

<u>clickCount:</u> Tracks the number of clicks.

3. Constructor Method



```java
    public leaguPanel() {
        initComponents();
        setOpaque(isOpaque: false);

        teamlist.addTeam(team: teampanel.team1);
        teamlist.addTeam(team: teampanel.team2);
        teamlist.addTeam(team: teampanel.team3);
        teamlist.addTeam(team: teampanel.team4);
        teamlist.addTeam(team: teampanel.team5);
        teamlist.addTeam(team: teampanel.team6);
        teamlist.addTeam(team: teampanel.team7);
        teamlist.addTeam(team: teampanel.team8);

        teampanel.team1.addPlayer(players:teampanel.playerList1);
        teampanel.team2.addPlayer(players:teampanel.playerList2);
        teampanel.team3.addPlayer(players:teampanel.playerList3);
        teampanel.team4.addPlayer(players:teampanel.playerList4);
        teampanel.team5.addPlayer(players:teampanel.playerList5);
        teampanel.team6.addPlayer(players:teampanel.playerList6);
        teampanel.team7.addPlayer(players:teampanel.playerList7);
        teampanel.team8.addPlayer(players:teampanel.playerList8);

        Node temp = teamlist.head;
        while (temp != null) {
            leagueTable.addRow(new Object[]{temp.team.name, temp.team.totalPoints, temp.team.scoreGoal, temp.team.concededGoal, 0, "-"});
            temp = temp.next;
        }
    }
```

initComponents(): A method used to initialize Swing components.

setOpaque(false): Makes the panel's background transparent.

Adding Teams : Adds eight teams defined in the teamPanel class to the teamlist data structure.

Adding Players: Adds players to each team using the addPlayer() method.

Creating the League Table:

teamlist.head: Refers to the first node in the linked list of teams.

Iterates through each team in the linked list:

Team name (name)-Total points (totalPoints)-Goals scored (scoreGoal)-Goals conceded (concededGoal)

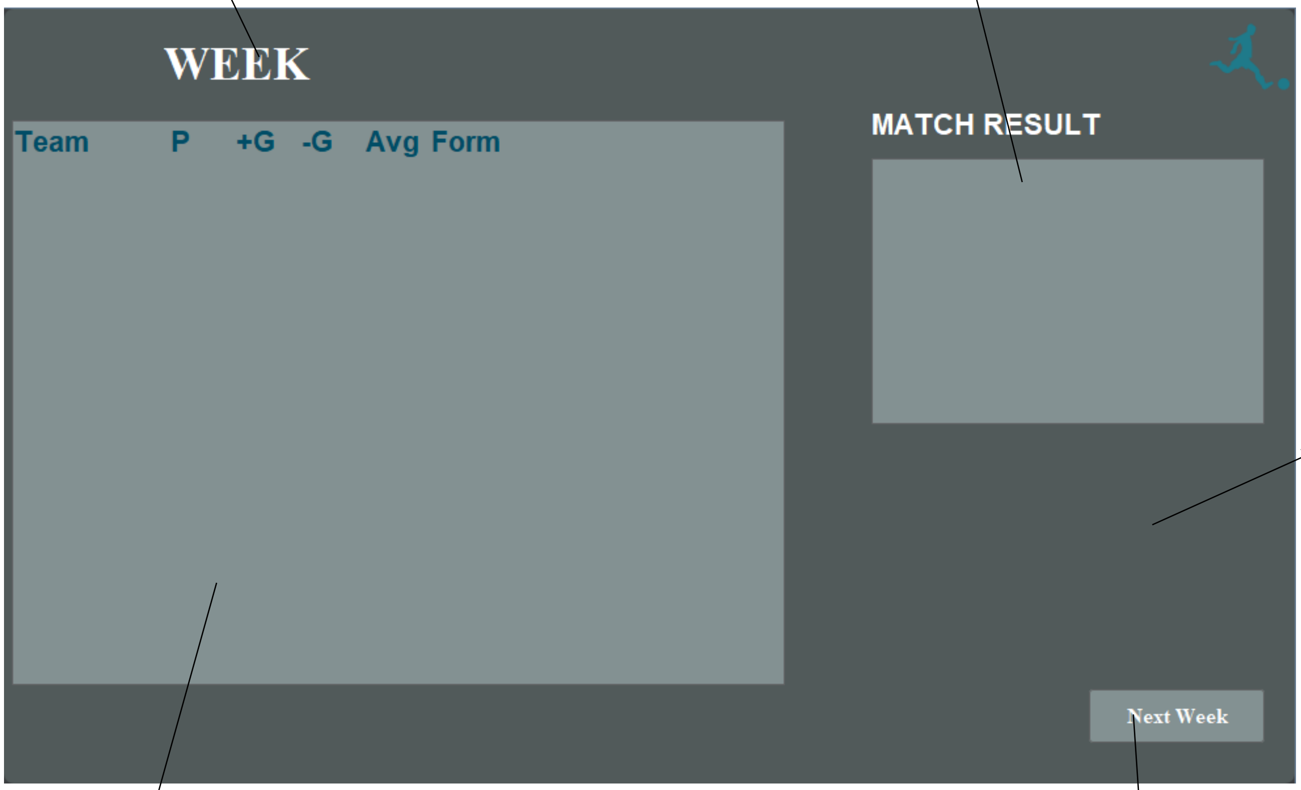
A fixed value (0)-A symbol ("-")

These details are added as a row to the leagueTable (likely a JTable object).

leaguepanel:

weekLabel: Shows which week the league is in

resultTable: The results of the week played are written in this table.

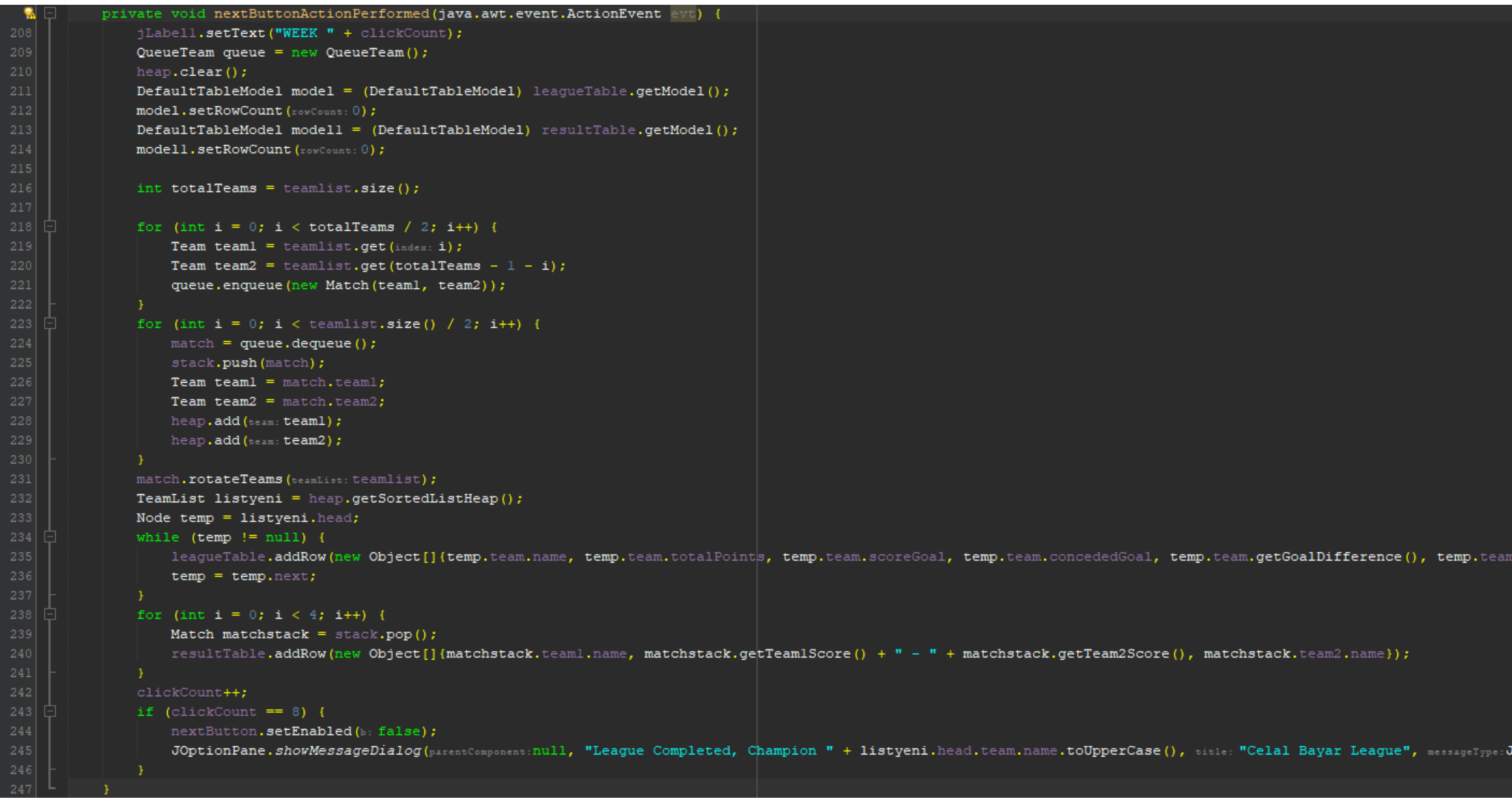


leaguepanel:

leagueTable: At the end of each week, teams are written into this table according to their points.

nextButton:

nextButton:



```java
    private void nextButtonActionPerformed(java.awt.event.ActionEvent evt) {
        jLabel1.setText("WEEK " + clickCount);
        QueueTeam queue = new QueueTeam();
        heap.clear();
        DefaultTableModel model = (DefaultTableModel) leagueTable.getModel();
        model.setRowCount(rowCount: 0);
        DefaultTableModel model1 = (DefaultTableModel) resultTable.getModel();
        model1.setRowCount(rowCount: 0);

        int totalTeams = teamlist.size();

        for (int i = 0; i < totalTeams / 2; i++) {
            Team team1 = teamlist.get(index: i);
            Team team2 = teamlist.get(totalTeams - 1 - i);
            queue.enqueue(new Match(team1, team2));
        }
        for (int i = 0; i < teamlist.size() / 2; i++) {
            match = queue.dequeue();
            stack.push(match);
            Team team1 = match.team1;
            Team team2 = match.team2;
            heap.add(team: team1);
            heap.add(team: team2);
        }
        match.rotateTeams(teamList: teamlist);
        TeamList listyeni = heap.getSortedListHeap();
        Node temp = listyeni.head;
        while (temp != null) {
            leagueTable.addRow(new Object[]{temp.team.name, temp.team.totalPoints, temp.team.scoreGoal, temp.team.concededGoal, temp.team.getGoalDifference(), temp.team
            temp = temp.next;
        }
        for (int i = 0; i < 4; i++) {
            Match matchstack = stack.pop();
            resultTable.addRow(new Object[]{matchstack.team1.name, matchstack.getTeam1Score() + " - " + matchstack.getTeam2Score(), matchstack.team2.name});
        }
        clickCount++;
        if (clickCount == 8) {
            nextButton.setEnabled(b: false);
            JOptionPane.showMessageDialog(parentComponent:null, "League Completed, Champion " + listyeni.head.team.name.toUpperCase(), title: "Celal Bayar League", messageType:J
        }
    }
```

This method is triggered when the Next Week button is clicked. It simulates a week in the league, processes match, updates standings, and advances the league to the next round.

Update Week Label: Updates jLabel1 to display the current week number using the clickCount variable.

Initialize Structures: Creates a new QueueTeam to manage match scheduling. Clear the heap to prepare for the current week's matches. Resets the league table (leagueTable) and match results table (resultTable).

Generate Matches: Matches are created using the first and last teams from the teamlist iteratively. These matches are added to the queue for processing.

Process Matches: Dequeues matches from the queue and processes each: Pushes matches to a stack for tracking. Adds teams involved in the match to the heap.

Rotate Teams: Teams are rotated using the match.rotateTeams() method for the next week's scheduling.

Update League Standings: Retrieves a sorted list of teams from the heap and updates the leagueTable with:

Team Name-Total Points-Goals Scored-Goals Conceded-Goal Difference-Form.

Display Match Results: Pops the last 4 matches from the stack and updates the resultTable with:

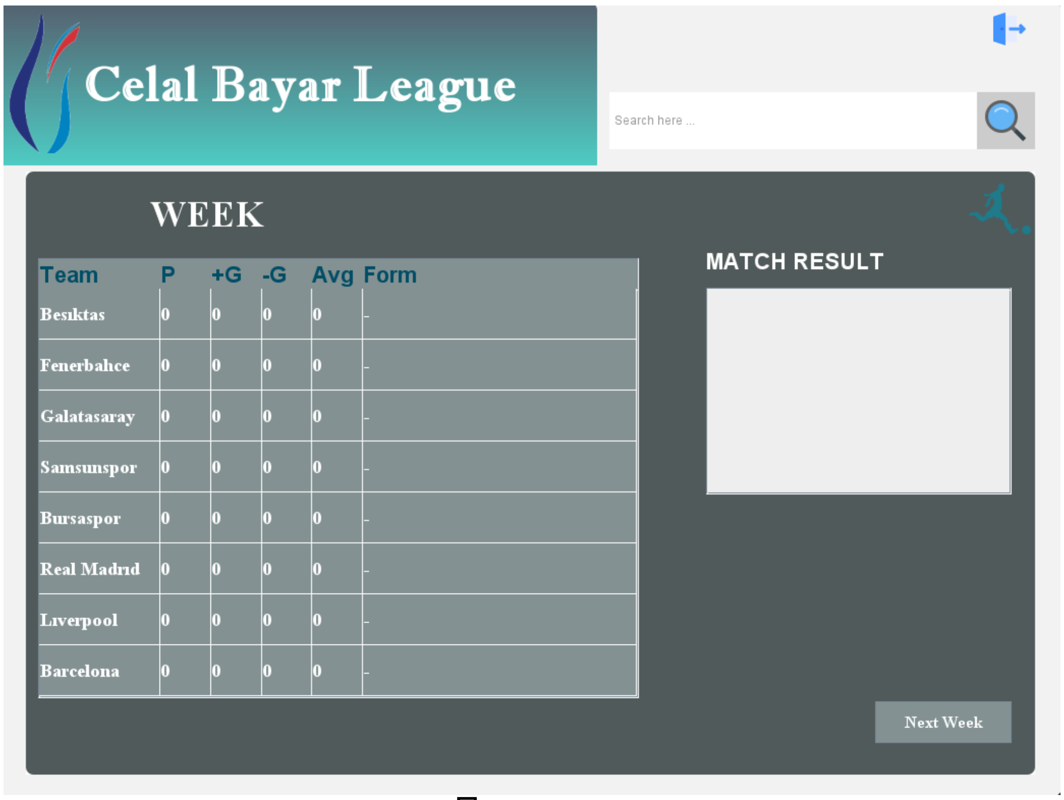
Team 1 Name-Match Score-Team 2 Name.

Advance Week and Check Completion: Increments clickCount. If all 7 weeks are completed:

Disables the "Next Week" button.

Displays the champion team in a message box.

OUTPUT

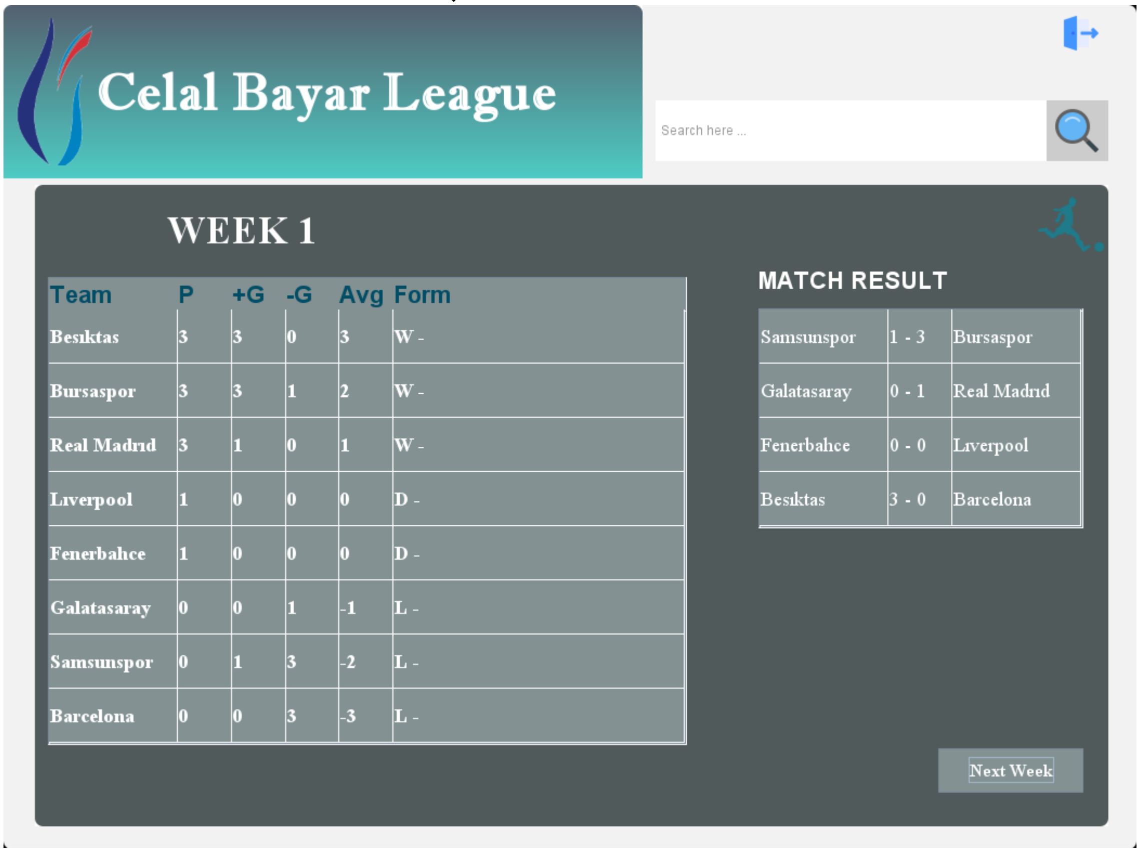


WEEK

Team	P	+G	-G	Avg	Form
Besıktas	0	0	0	0	-
Fenerbahce	0	0	0	0	-
Galatasaray	0	0	0	0	-
Samsunspor	0	0	0	0	-
Bursaspor	0	0	0	0	-
Real Madrıd	0	0	0	0	-
Lıverpool	0	0	0	0	-
Barcelona	0	0	0	0	-

MATCH RESULT

Click the Next Week button.



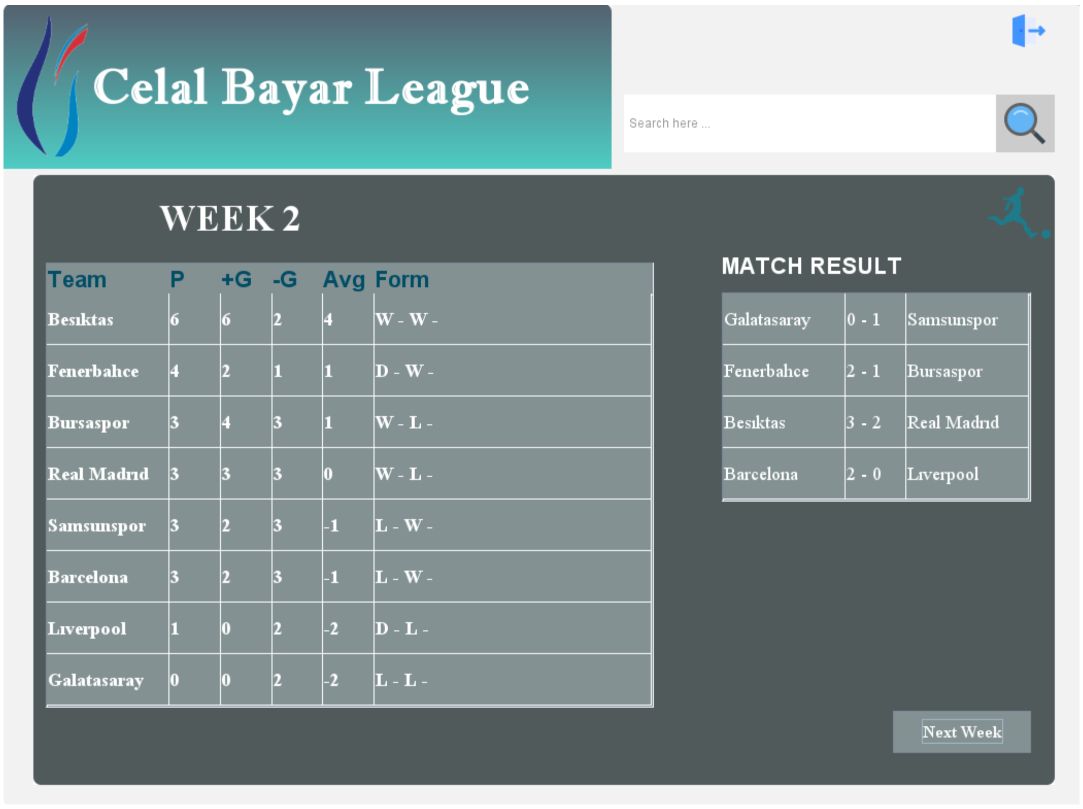
WEEK 1

Team	P	+G	-G	Avg	Form
Besıktas	3	3	0	3	W -
Bursaspor	3	3	1	2	W -
Real Madrıd	3	1	0	1	W -
Lıverpool	1	0	0	0	D -
Fenerbahce	1	0	0	0	D -
Galatasaray	0	0	1	-1	L -
Samsunspor	0	1	3	-2	L -
Barcelona	0	0	3	-3	L -

MATCH RESULT

Samsunspor	1 - 3	Bursaspor
Galatasaray	0 - 1	Real Madrıd
Fenerbahce	0 - 0	Lıverpool
Besıktas	3 - 0	Barcelona

Click the Next Week button.



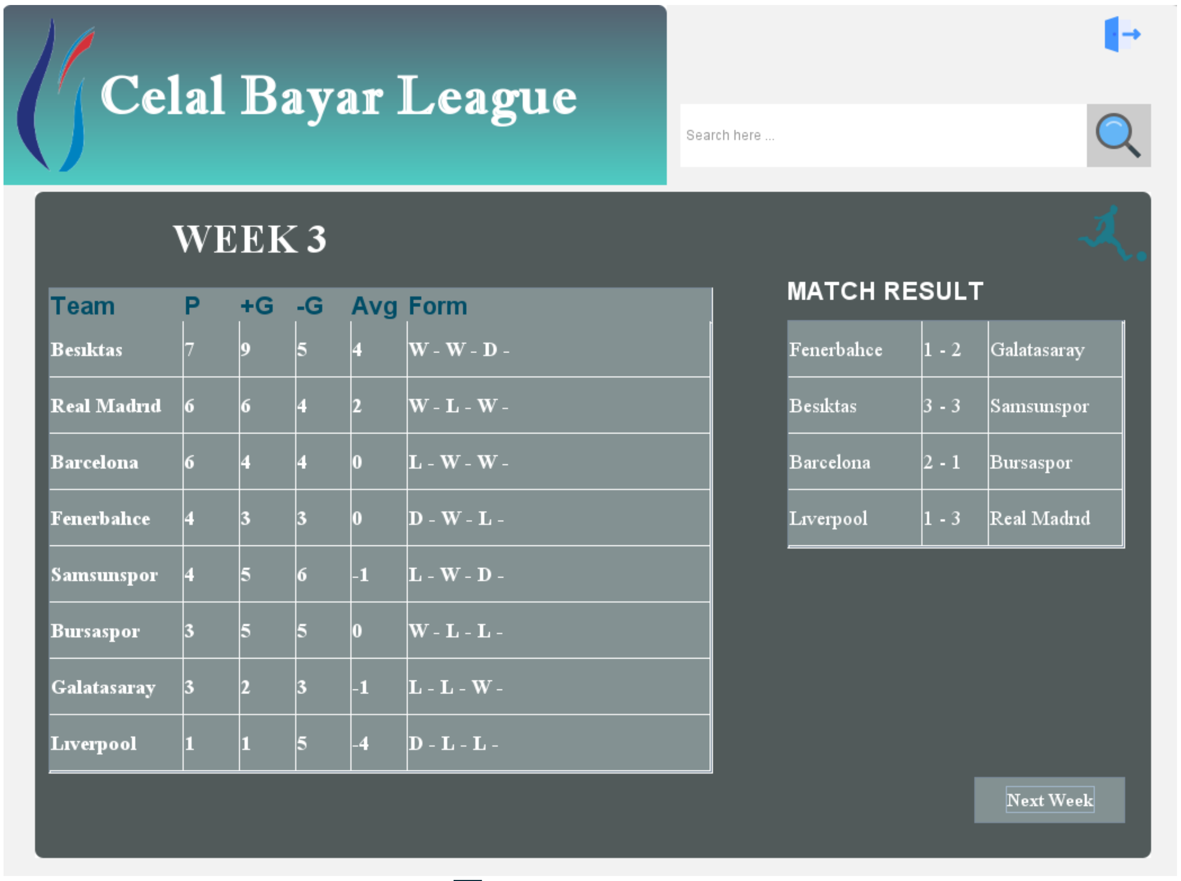
WEEK 2

Team	P	+G	-G	Avg	Form
Besıktas	6	6	2	4	W - W -
Fenerbahce	4	2	1	1	D - W -
Bursaspor	3	4	3	1	W - L -
Real Madrıd	3	3	3	0	W - L -
Samsunspor	3	2	3	-1	L - W -
Barcelona	3	2	3	-1	L - W -
Lıverpool	1	0	2	-2	D - L -
Galatasaray	0	0	2	-2	L - L -

MATCH RESULT

Galatasaray	0 - 1	Samsunspor
Fenerbahce	2 - 1	Bursaspor
Besıktas	3 - 2	Real Madrıd
Barcelona	2 - 0	Lıverpool

Click the Next Week button.



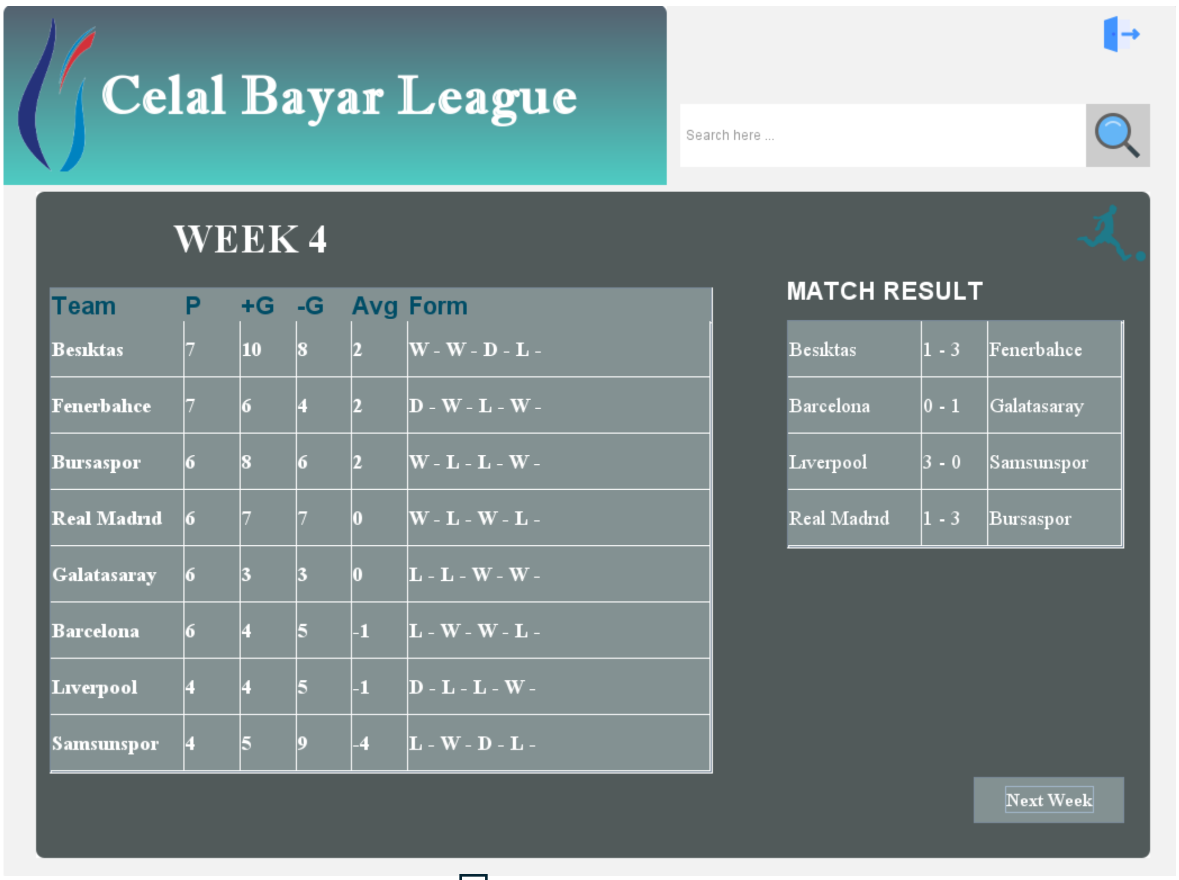
WEEK 3

Team	P	+G	-G	Avg	Form
Besıktas	7	9	5	4	W - W - D -
Real Madrıd	6	6	4	2	W - L - W -
Barcelona	6	4	4	0	L - W - W -
Fenerbahce	4	3	3	0	D - W - L -
Samsunspor	4	5	6	-1	L - W - D -
Bursaspor	3	5	5	0	W - L - L -
Galatasaray	3	2	3	-1	L - L - W -
Lıverpool	1	1	5	-4	D - L - L -

MATCH RESULT

Fenerbahce	1 - 2	Galatasaray
Besıktas	3 - 3	Samsunspor
Barcelona	2 - 1	Bursaspor
Lıverpool	1 - 3	Real Madrıd

Click the Next Week button.



Celal Bayar League

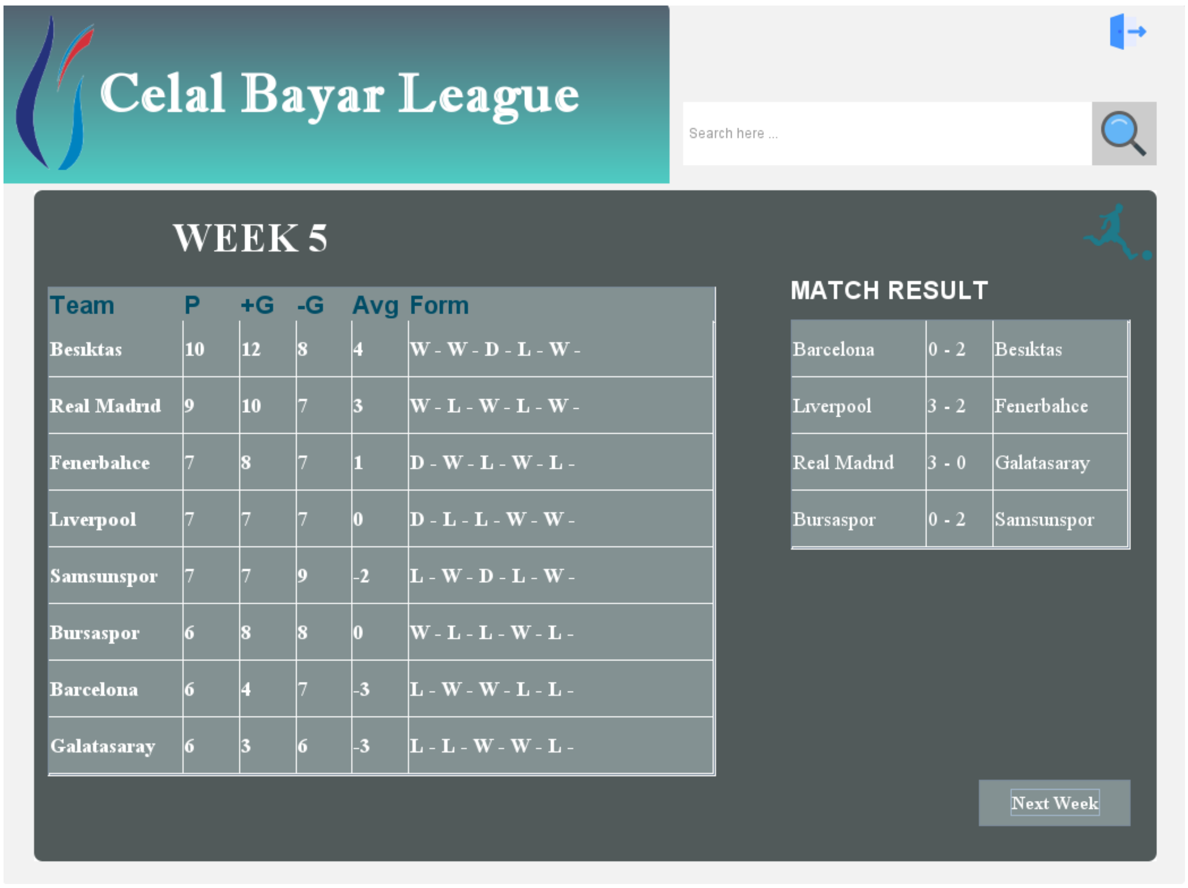
WEEK 4

Team	P	+G	-G	Avg	Form
Besıktas	7	10	8	2	W - W - D - L -
Fenerbahce	7	6	4	2	D - W - L - W -
Bursaspor	6	8	6	2	W - L - L - W -
Real Madrıd	6	7	7	0	W - L - W - L -
Galatasaray	6	3	3	0	L - L - W - W -
Barcelona	6	4	5	-1	L - W - W - L -
Lıverpool	4	4	5	-1	D - L - L - W -
Samsunspor	4	5	9	-4	L - W - D - L -

MATCH RESULT

Besıktas	1 - 3	Fenerbahce
Barcelona	0 - 1	Galatasaray
Lıverpool	3 - 0	Samsunspor
Real Madrıd	1 - 3	Bursaspor

Click the Next Week button.



Celal Bayar League

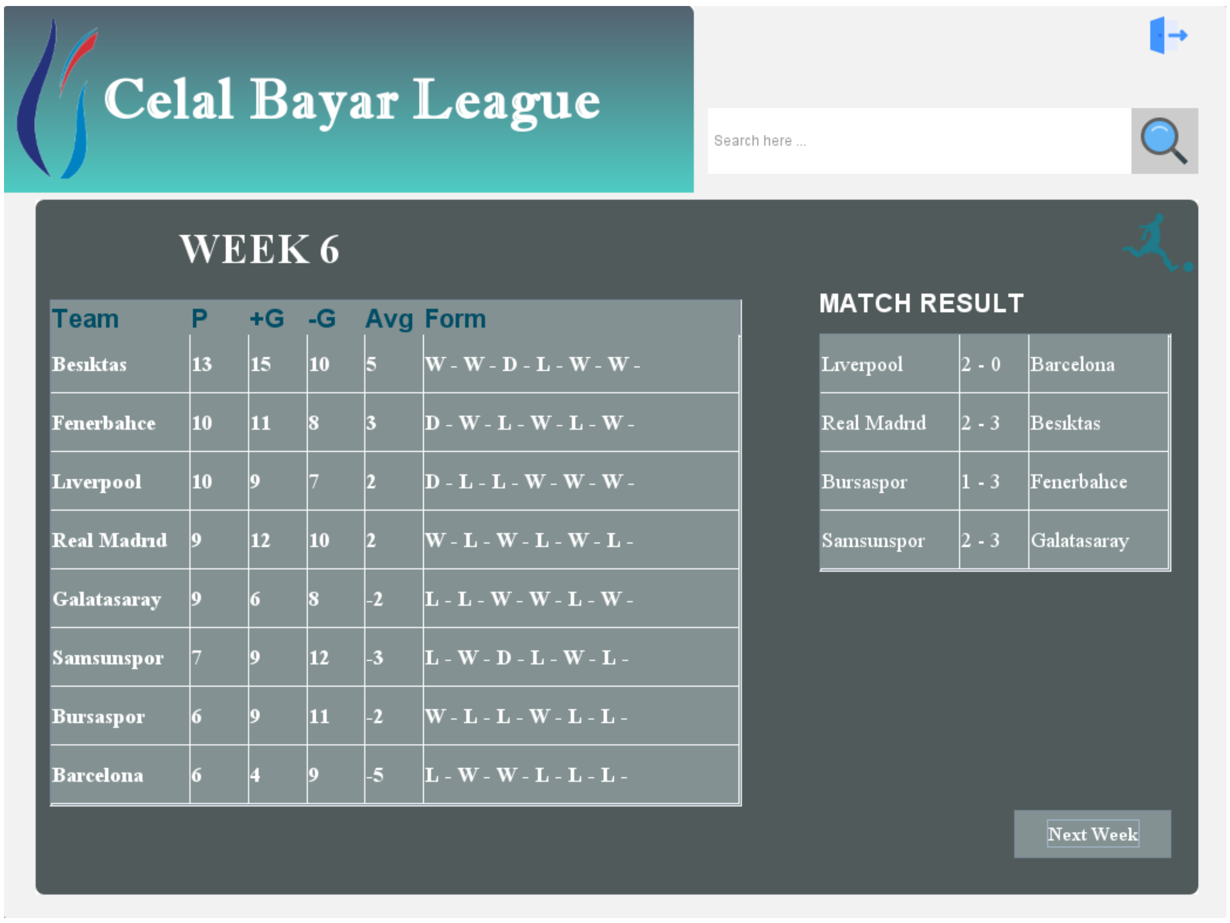
WEEK 5

Team	P	+G	-G	Avg	Form
Besıktas	10	12	8	4	W - W - D - L - W -
Real Madrıd	9	10	7	3	W - L - W - L - W -
Fenerbahce	7	8	7	1	D - W - L - W - L -
Lıverpool	7	7	7	0	D - L - L - W - W -
Samsunspor	7	7	9	-2	L - W - D - L - W -
Bursaspor	6	8	8	0	W - L - L - W - L -
Barcelona	6	4	7	-3	L - W - W - L - L -
Galatasaray	6	3	6	-3	L - L - W - W - L -

MATCH RESULT

Barcelona	0 - 2	Besıktas
Lıverpool	3 - 2	Fenerbahce
Real Madrıd	3 - 0	Galatasaray
Bursaspor	0 - 2	Samsunspor

Click the Next Week button.


Celal Bayar League

Search here ...

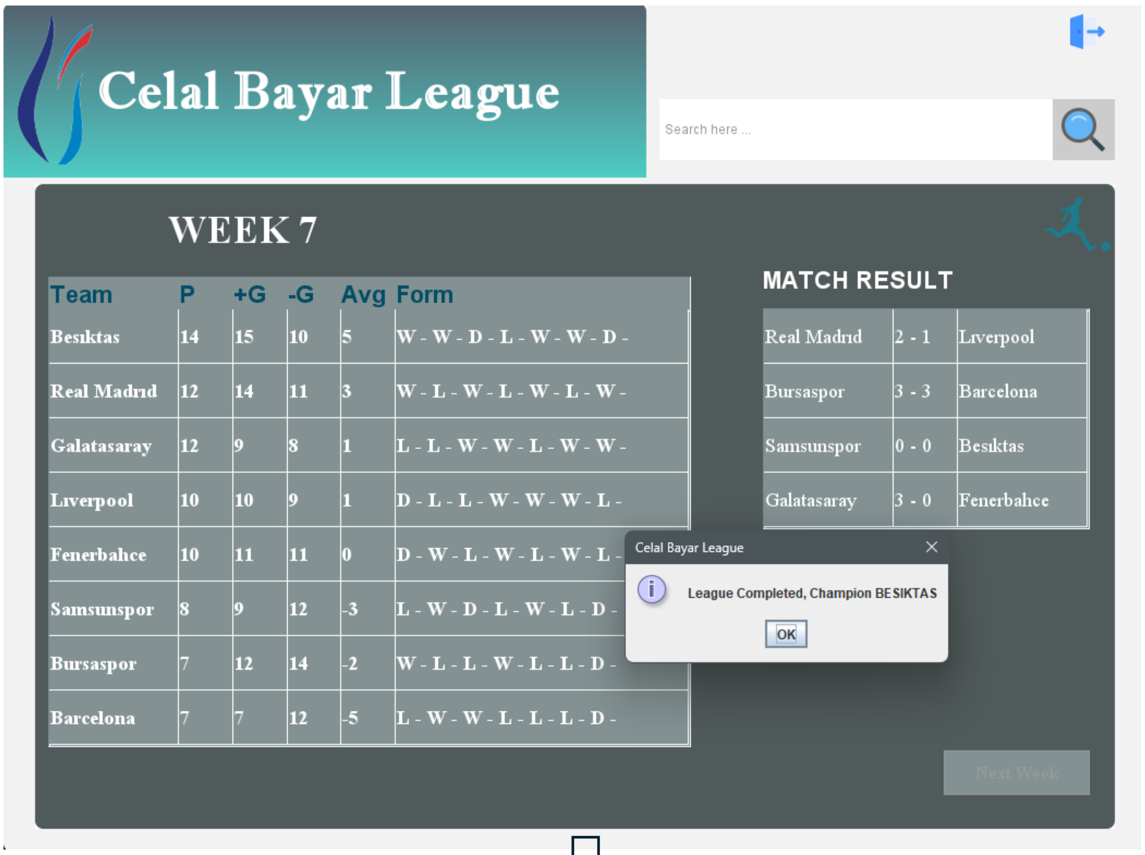
WEEK 6

Team	P	+G	-G	Avg	Form
Besıktas	13	15	10	5	W - W - D - L - W - W -
Fenerbahce	10	11	8	3	D - W - L - W - L - W -
Lıverpool	10	9	7	2	D - L - L - W - W - W -
Real Madrıd	9	12	10	2	W - L - W - L - W - L -
Galatasaray	9	6	8	-2	L - L - W - W - L - W -
Samsunspor	7	9	12	-3	L - W - D - L - W - L -
Bursaspor	6	9	11	-2	W - L - L - W - L - L -
Barcelona	6	4	9	-5	L - W - W - L - L - L -

MATCH RESULT

Lıverpool	2 - 0	Barcelona
Real Madrıd	2 - 3	Besıktas
Bursaspor	1 - 3	Fenerbahce
Samsunspor	2 - 3	Galatasaray

Click the Next Week button.


Celal Bayar League

Search here ...

WEEK 7

Team	P	+G	-G	Avg	Form
Besıktas	14	15	10	5	W - W - D - L - W - W - D -
Real Madrıd	12	14	11	3	W - L - W - L - W - L - W -
Galatasaray	12	9	8	1	L - L - W - W - L - W - W -
Lıverpool	10	10	9	1	D - L - L - W - W - W - L -
Fenerbahce	10	11	11	0	D - W - L - W - L - W - L -
Samsunspor	8	9	12	-3	L - W - D - L - W - L - D -
Bursaspor	7	12	14	-2	W - L - L - W - L - L - D -
Barcelona	7	7	12	-5	L - W - W - L - L - L - D -

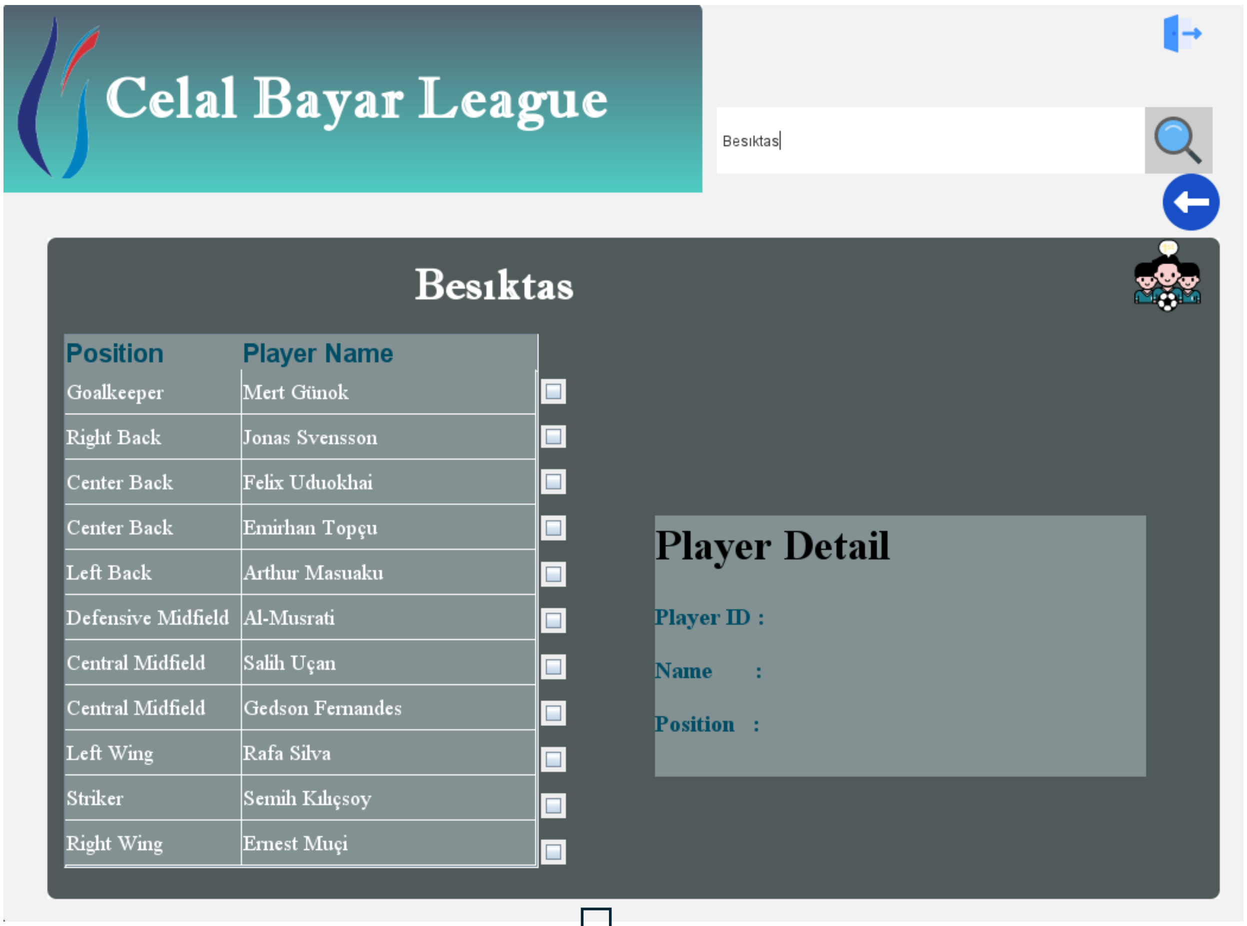
MATCH RESULT

Real Madrıd	2 - 1	Lıverpool
Bursaspor	3 - 3	Barcelona
Samsunspor	0 - 0	Besıktas
Galatasaray	3 - 0	Fenerbahce

Celal Bayar League

League Completed, Champion BESIKTAS

Search Besıktas and click the button.



Besıktas

Position	Player Name	
Goalkeeper	Mert Günok	☐
Right Back	Jonas Svensson	☐
Center Back	Felix Uduokhai	☐
Center Back	Emirhan Topçu	☐
Left Back	Arthur Masuaku	☐
Defensive Midfield	Al-Musrati	☐
Central Midfield	Salih Uçan	☐
Central Midfield	Gedson Fernandes	☐
Left Wing	Rafa Silva	☐
Striker	Semih Kılıçsoy	☐
Right Wing	Ernest Muçi	☐

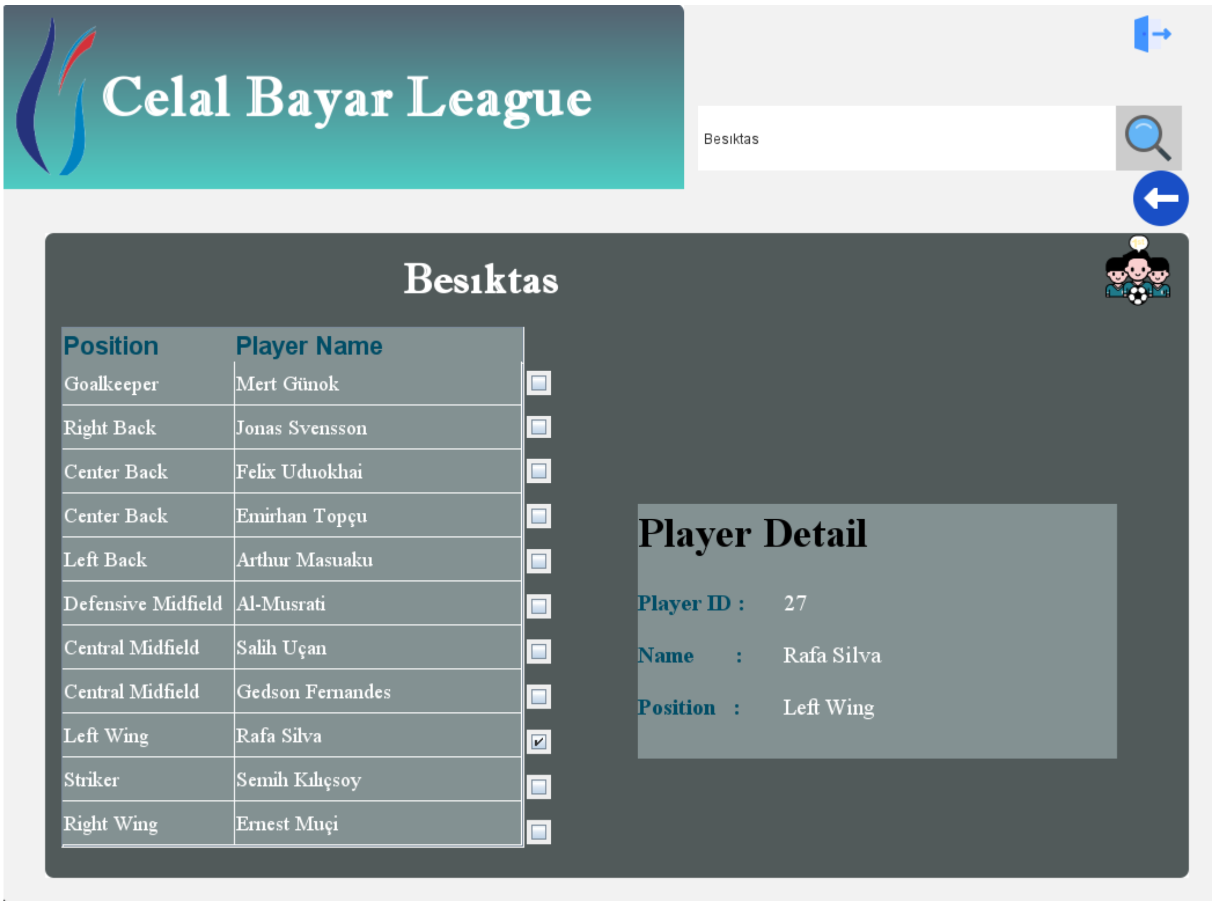
Player Detail

Player ID :

Name :

Position :

Click check box the player you want.



Besıktas

Position	Player Name	
Goalkeeper	Mert Günok	☐
Right Back	Jonas Svensson	☐
Center Back	Felix Uduokhai	☐
Center Back	Emirhan Topçu	☐
Left Back	Arthur Masuaku	☐
Defensive Midfield	Al-Musrati	☐
Central Midfield	Salih Uçan	☐
Central Midfield	Gedson Fernandes	☐
Left Wing	Rafa Silva	☑
Striker	Semih Kılıçsoy	☐
Right Wing	Ernest Muçi	☐

Player Detail

Player ID : 27

Name : Rafa Silva

Position : Left Wing

Click the exit button.