**Dokumentacja techniczna gry**

W tym miejscu będziemy prezentować programy stworzone na potrzeby projektowania gry, wraz z szczegółowym wyjaśnieniem ich działania. Każdy z programów będzie omówiony pod kątem jego funkcjonalności i wykorzystania w grze.

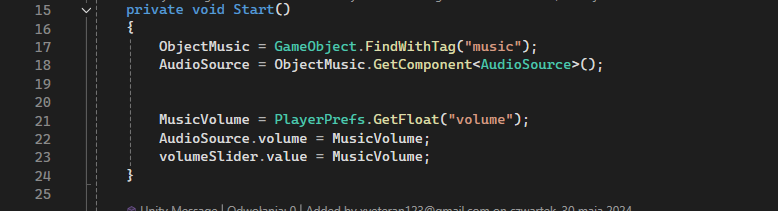
Kolejność omawianych programów jest losowa

MusicPlayerScript.cs – jest to skrypt, który ma za zadanie zarządzać odtwarzaniem muzyki i regulacją dźwięku w grze.

Ustawia dźwięk na wartość domyślną



Start:



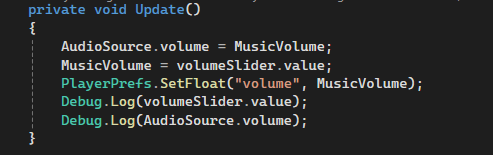
ObjectMusic – znajduje obiekt z tagiem music i przypisuje go do zmiennej.

AudioSource – pobiera utworzony w grze komponent „AudioSource” i przypisuje go do zmiennej

PlayerPrefs – odpowiada za zapisywanie wartości dźwięku w pramięci gry.

AudioSource.volume i volumeSlider.value – ustawine są wartości na bazie zmiennej MusicVolume

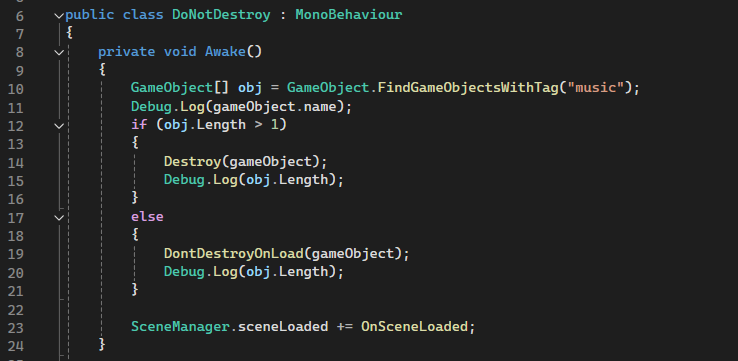
Update:



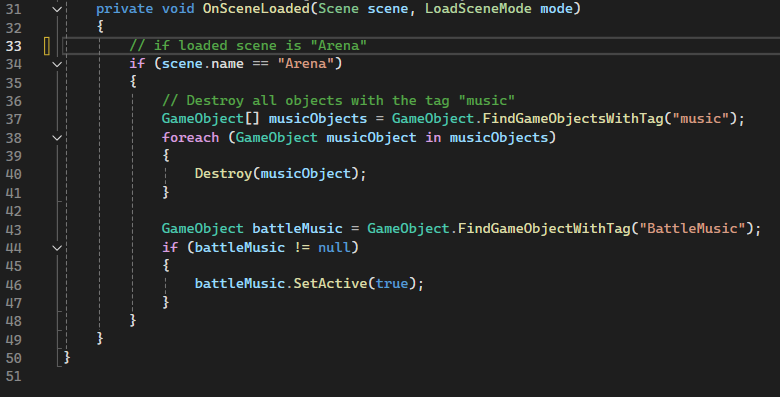
Aktualizacja głośności "AudioSource" na podstawie wartości "MusicVolume", która synchronizuje wartość suwaka "volumeSlider".   
  
Następnie zapisuje tę wartość do PlayerPrefs(czyli, jak zresetujemy grę, wartość będzie taka jak poprzednio), a debug, służyły do śledzenia wartości (podczas testów)

Kiedy była już obsługa suwaka, potrzebny był skrypt, który pozwoliłby odtwarzać muzykę pomiędzy różnymi scenami (i żeby utwór nie uruchamiał się przy przejściu na kolejną scenę od nowa)

# DoNotDestoy.cs

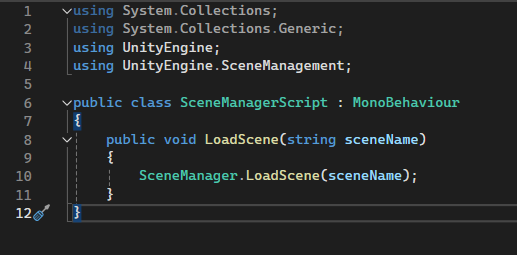


Ten skrypt ma zapobiegać „niszczeniu”(niszczenie, ogólnie wywołuje się przy przejściu na kolejną scenę, czyli żaden obiekt z poprzedniej sceny nie przenosi się do kolejnej) obiektu z tagiem „music” przy zmianie sceny, o ile nie istnieje już obiekt z takim tagiem. Jeżeli pojawi się obiekt nadmiarowy, to zostanie zniszczony, jeżeli jest tylko jeden, zostaje on zachowany przy zmianie sceny.



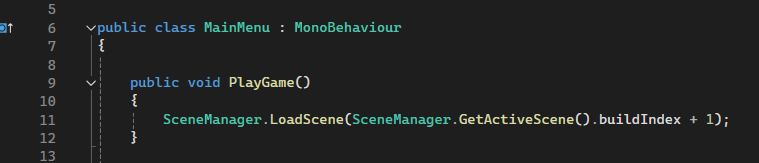
Ten fragment jest wykorzystywany do zrealizowania muzyki na scenie Areny, sprawdza scenę, jeżeli jest Areną, usuwa obiekt, który ma tag = „music”, następnie szuka, czy w danej scenie, jakiś obiekt ma tag „BattleMusic”, jeśli tak to uruchamia go.

# LoadNextSceneScript.cs



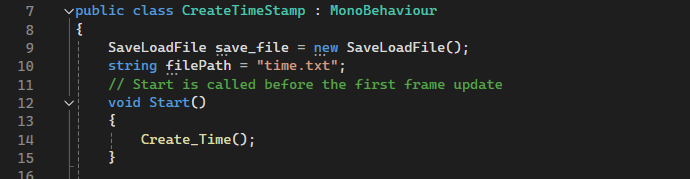
Bardzo prosty skrypt, który ma za zadanie, uruchomić konkretną scenę, np. kiedy przypisujemy akcje do przycisku (weaponsmith, arena itp.)

# MainMenu.cs

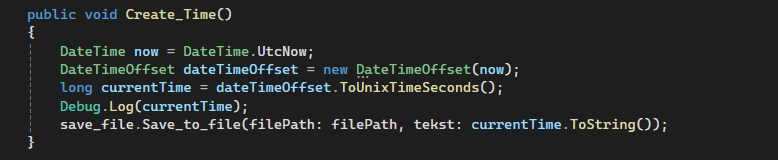


Skrypt, który bardzo szybko został zastąpiony przez LoadNextSceneScript.cs wykorzystywany był w początkowej fazie projektu, do przechodzenia do następnej sceny

CreateTimeStamp.cs – skrypt który ma za zadanie mierzyć czas gry



Ten fragment kodu tworzy znacznik czasu przy uruchomieniu obiektu w grze i zapisuje go do pliki time.txt

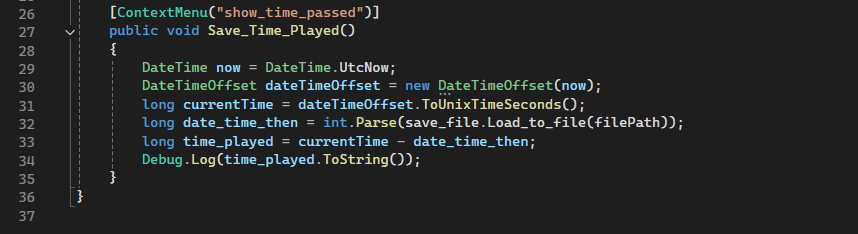


Druga część kodu pokazuje implemetracje metody Create\_time

„DateTime now” – zapisuje do zmiennej aktualny czas

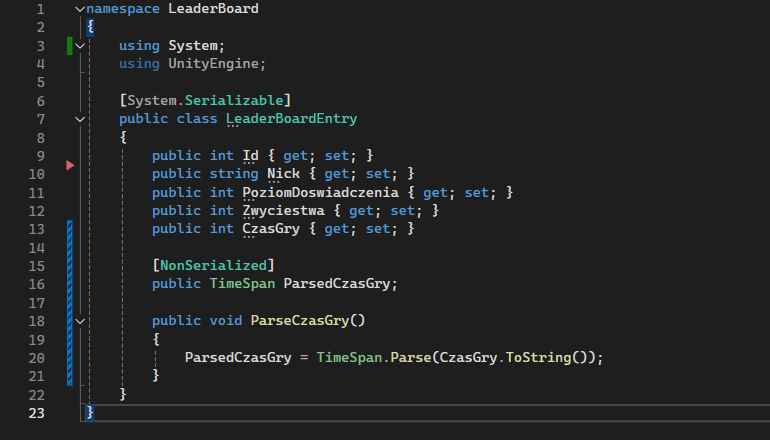
long currentTime = dateTimeOffset.ToUnixTimeSeconds(); - polega na konwertowaniu czasu na liczbę sekund

ostatnia linijka z tego fragmentu kodu pokazuje zapisanie czasu do pliku tekstowego



Ostatnia część tego skryptu pozwala na zapisanie znacznika czasu i późniejsze obliczenie oraz wyświetlanie czasu który upłynął od momentu tego zapisu.

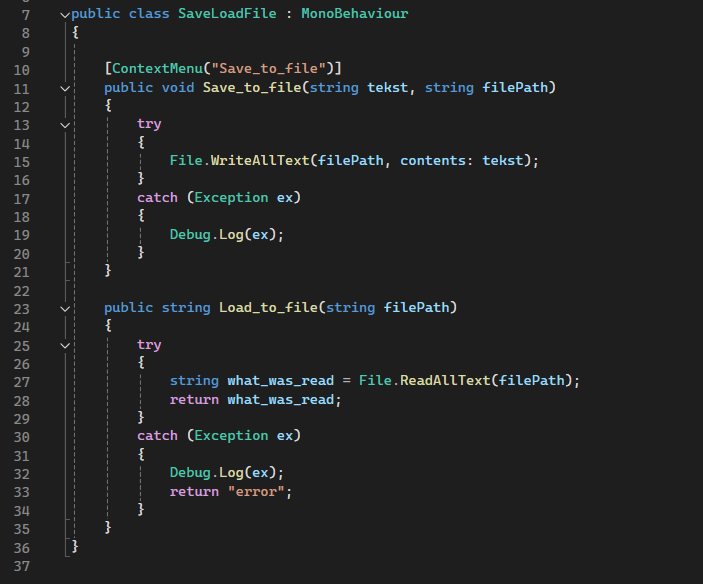
# LeaderBoard.cs



Ten skrypt opisuje strukturę tabeli w bazie danych

Warto zwrócić uwagę na pola: [Serializable] i [NonSerialized] są to pola które mają lub nie mają być przechwycone/przesyłane przez sieć.

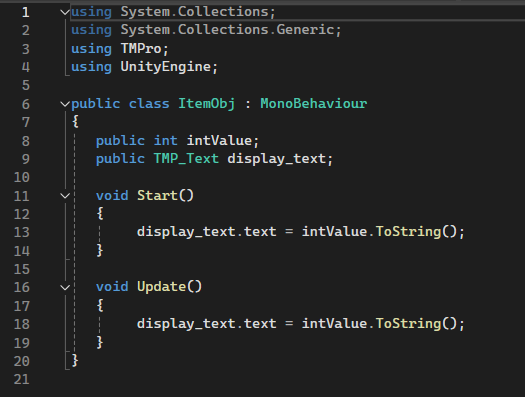
# SaveLoadFile.cs



Save\_to\_file – zapisuje tekst do określonego pliku   
Load\_to\_file – odczytuje tekst z określonego pliku

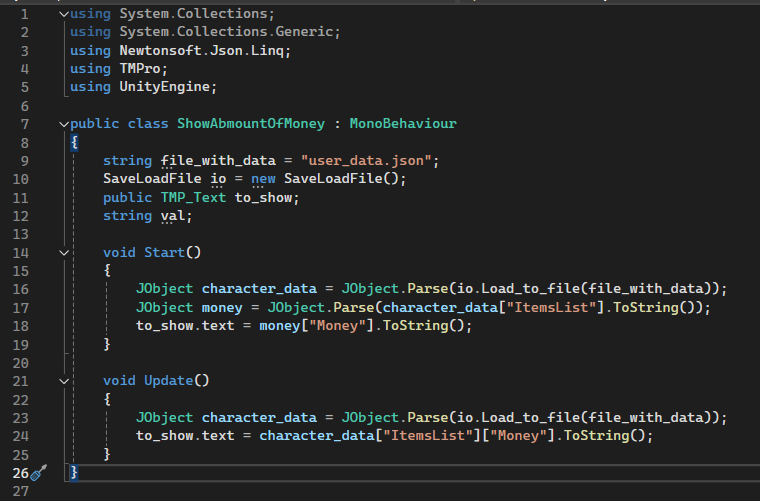
Skrypt jest wykorzystywany do interakcji z plikami podczas gry

# ItemObj.cs



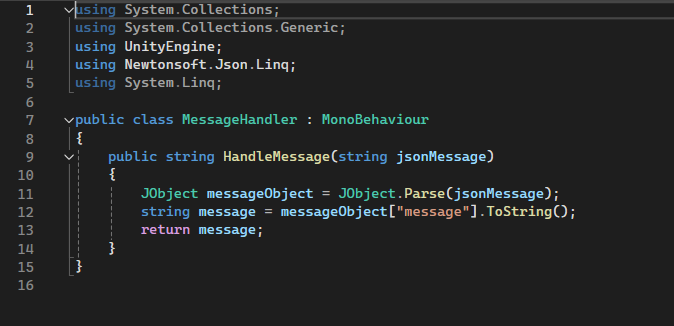
Skrypt ten ma za zadanie ustawić wartość pola tekstowego, kiedy obiekt zaczyna być aktywny i aktualizować go kiedy pojawiają się zmiany (oczywiście taką jaką przyjmuje wartość intValue)

# ShowMoney.cs



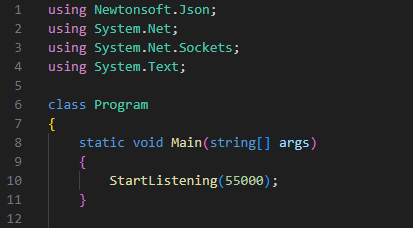
Skrypt odczytuje wartość z pliku user\_data.json i wyświetla walutę użytkownika na ekranie za pomocą TextMesh Pro (pola tekstowego w Unity)

# MessageHandler.cs

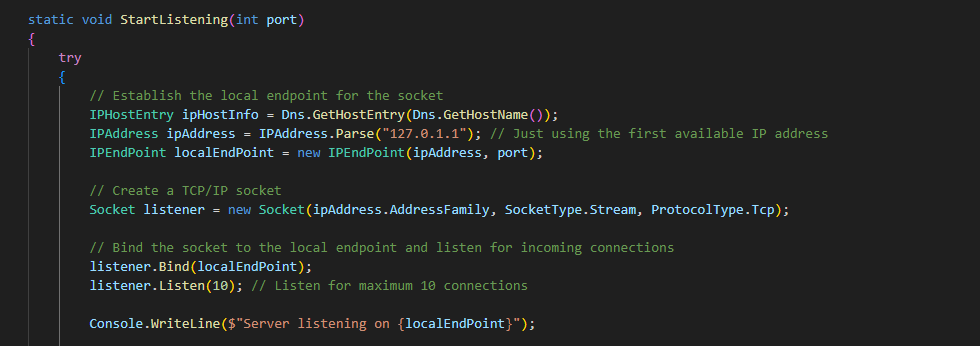


Ten skrypt przyjmuje ciąg znaków w formacie json, parsuje ten ciąg do JObject, wyciąga wartość z pola „message” i zwraca wartość jako ciąg znaków

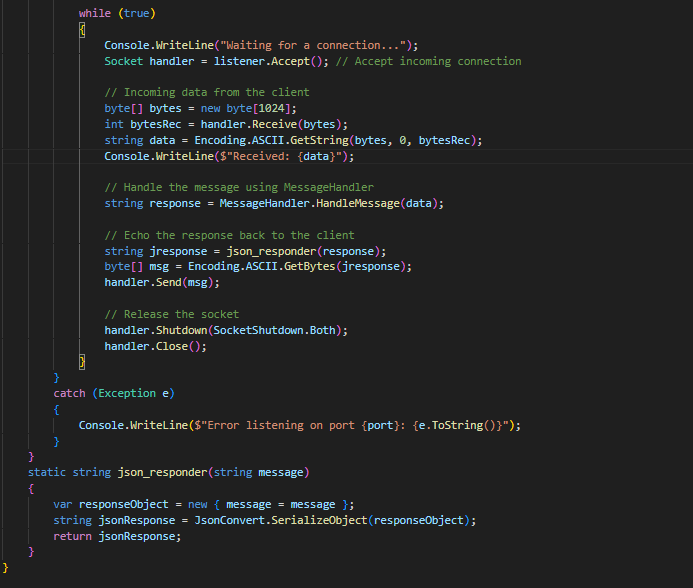
# Serwer: Program.cs



Metoda „Main” uruchamia nasłuchiwanie na porcie: 55000



Kiedy mamy już ustalony port, definiuje lokalny adres IP „127.0.1.1” do nasłuchiwania połączeń, następnie tworzy socet tcp/ip i powiązuje je z adresem IP i portem. Później ustala, że maksymalna ilość połączeń jest równa 10.



W tej pętli serwer oczekuje na połączenie się klienta, kiedy połączenie zostanie zaakceptowane, odbiera dane od klienta i zapisuje w buforze. Przy użyciu MessageHandlera przetwarza te dane. Później tworzy odpowiedź w formacie json przy użyciu metody „json\_responder”. Następnie wysyła odpowiedź do klienta i zamyka połączenie w obu kierunkach, jeżeli w trakcie tego procesu wystąpi jakiś błąd, zostanie przechwycony przez obsługę błędów i go wypisze.

Handler.cs – jest to kod, który ma za zadanie obsługiwać różne rodzaje wiadomości przesłanych do serwera. Przyjmuje wiadomości, analizuje którą pętle spełnia i zwraca odpowiedź.

Rozpatrzone jest wiele możliwości takich jak m.in odczytywanie użytkowników z bazy danych, sprawdzanie czy użytkownik istnieje w tej bazie danych, sprawdza czy istnieje już nickname postaci. Handler w tym samym kodzie, również obsługuje przypadki w których klient wysyła dane do serwera/bazy danych m.in. dodanie nowej postaci do bazy danych, dodanie nowych danych logowania użytkownika do bazy danych

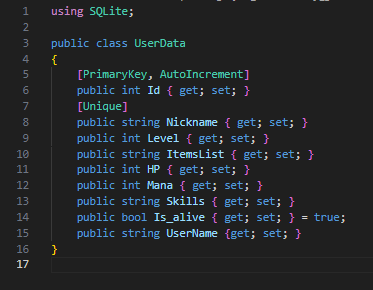
Obsługa błędów jest również zdefiniowana, czyli jeżeli nie będzie w stanie przetworzyć danej wiadomości, zwróci komunikat o błędzie

Serwer/db:

Tutaj będzie będą pokazane wszystkie funkcje związane operacjami na bazie danych.

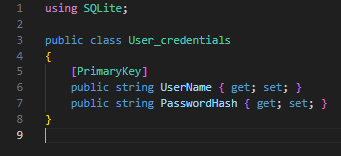
Definiowanie tabeli w bazie danych:

# User\_data.cs



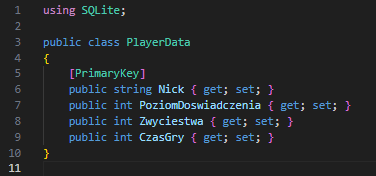
Ten kod definiuje tabele w których w celu przechowywania danych użytkowników.

# User\_credentials.cs



Ten kod definiuje tabelę z nazwą użytkownika(hashowany) i hasłem użytkownika(również hashowanym)

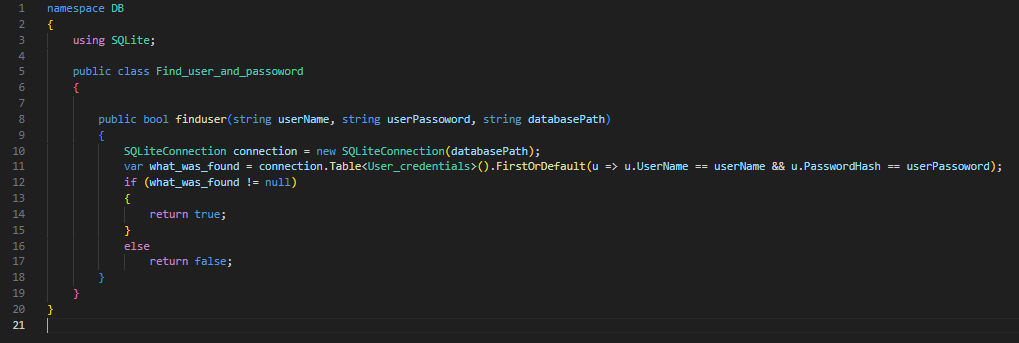
# Score\_scructur.cs



Ten kod definiuje tabelę, która jest wykorzystywana do leaderboard’a

Wyszukiwanie użytkownika w bazie danych

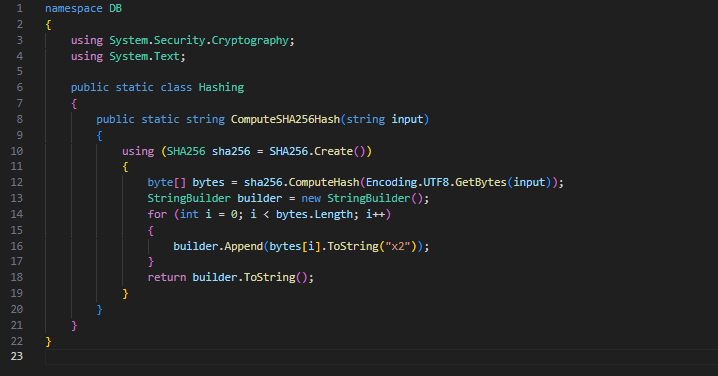
# Find\_user\_cred.cs



Ten kod jest używany do sprawdzania, czy użytkownik i hasło istnieje w bazie danych. Metoda finduser łączy się z bazą danych i przeszukuje tabelę „Users\_credentials” w poszukiwaniu pasującego użytkownika i zwraca „true” albo „false”

Hashowanie

# Hashing.cs



Ten kod ma za zadanie przekształcić dane wpisane przez użytkownika na SHA-256 w celu bezpiecznego przychowania tych danych w bazie danych.

Odczytywanie danych

# Reader.cs

Zawiera metody do odczytywania danych z bazy danych, a następnie zwracane są jako json, co ułatwia dalsze przetwarzanie i wyświetlanie tych danych.

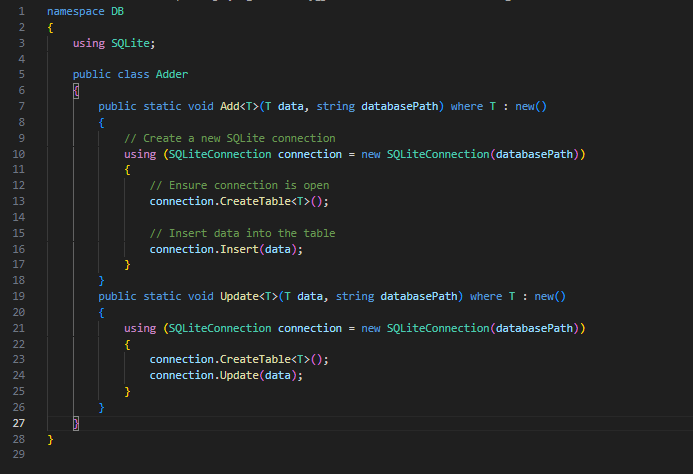
Read.scoreboard – odczytuje dane wszystkich graczy i zwraca listę (nick, poziom doświadczenia, zwycięstwa i czas gry)

Leaderboard\_entry – przeszukuje bazę danych, aby znaleźć konkretnego gracza po jego nicku i zwraca jego dane

Read\_user\_data – odczytuje wszystkie dane gracza i ponownie zwraca listę (ID, nickname, level, itemlist…)

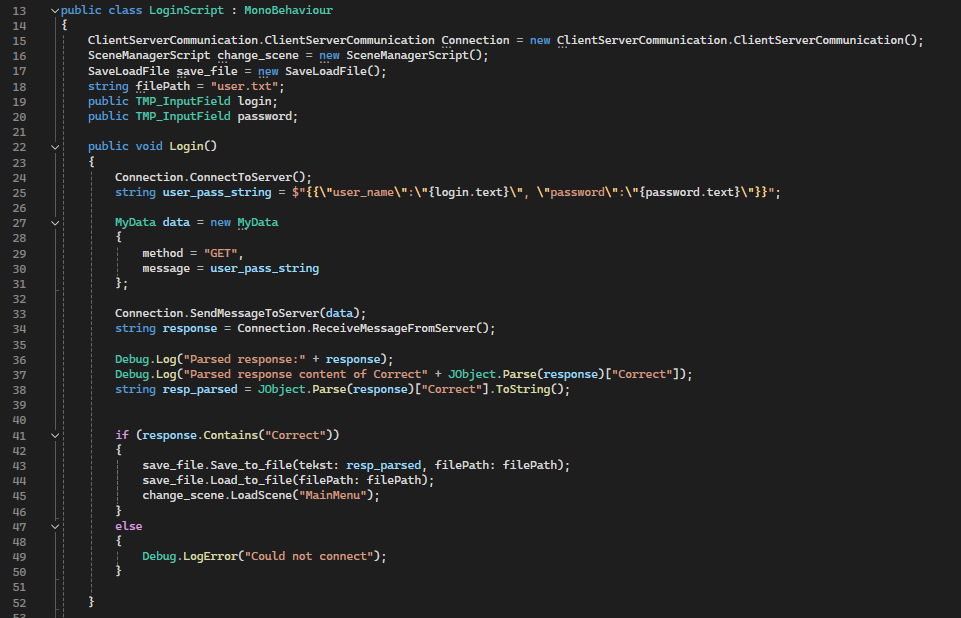
Read\_user\_info – szuka w bazie konkretnego gracza po jego nazwie użytkownika, jeżeli dane są to zwraca je, jeżeli nie wyświetla komunikat o braku danych.

Dodawanie i aktualizowanie danych



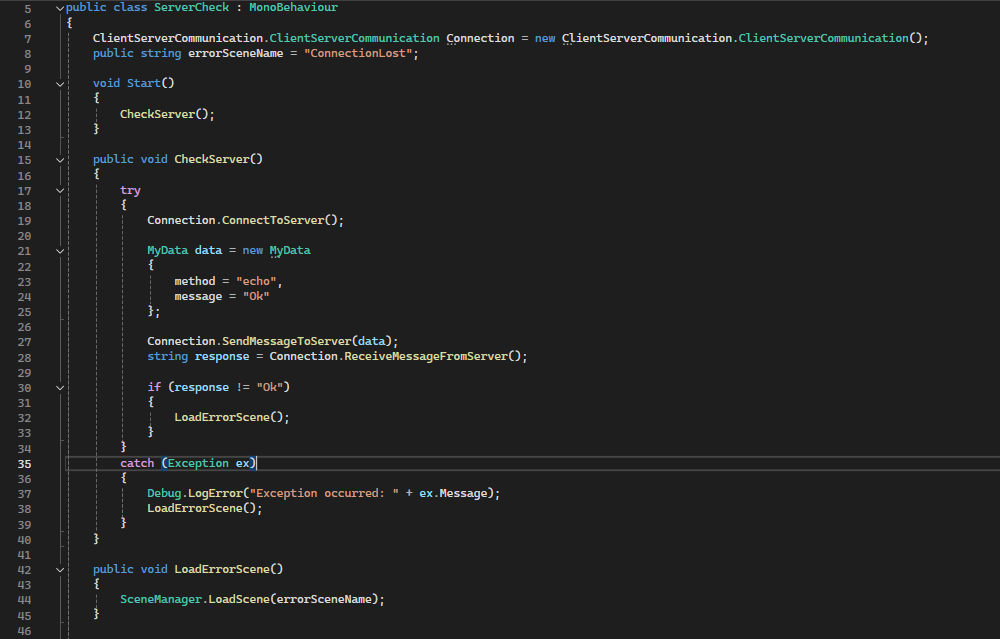
Dwie metody, które są używane w tym kodzie są używane przy dodawaniu i aktualizowaniu rekordów w bazie danych.

# LoginScript.cs



Ten skrypt ma za zadanie zarządzać procesem logowania się użytkownika w grze, na początku pobiera od niego daneg logowania i wysyła je do serwera, następnie odbiera odpowiedź i w zależności od niej podejmuje decyzje, takie jak zapis danych użytkownika do pliku i przejście na kolejną scenę.

# ServerCheck.cs

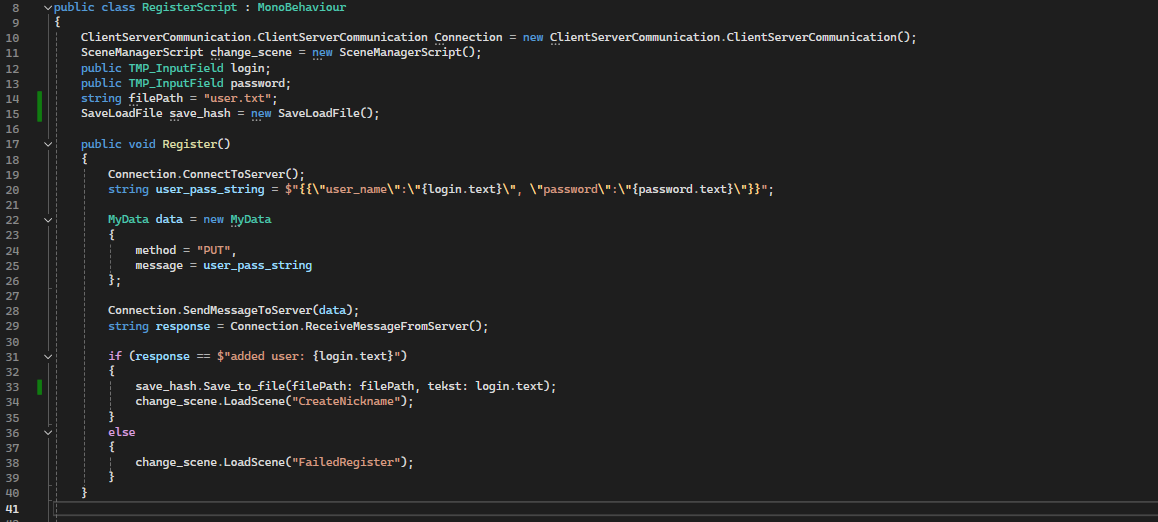


Jest to skrypt który ma za zadanie sprawdzić, czy aplikacja może połączyć się z serwerem. Na początku działania aplikacji próbuje nawiązać połączenie z serwerem i wysyła prostą wiadomość „echo”. Jeśli serwer odpowie poprawnie, aplikacja kontynuuje działanie, jeżeli natomiast nie odpowie, lub wystąpi błąd, zostaje załadowana scena błędu, która informuje użytkownika o błędzie.

Ten skrypt, działa cały czas, na każdej scenie, sprawdzając czy w danym momencie gry, nie doszło do problemu z komunikacją.

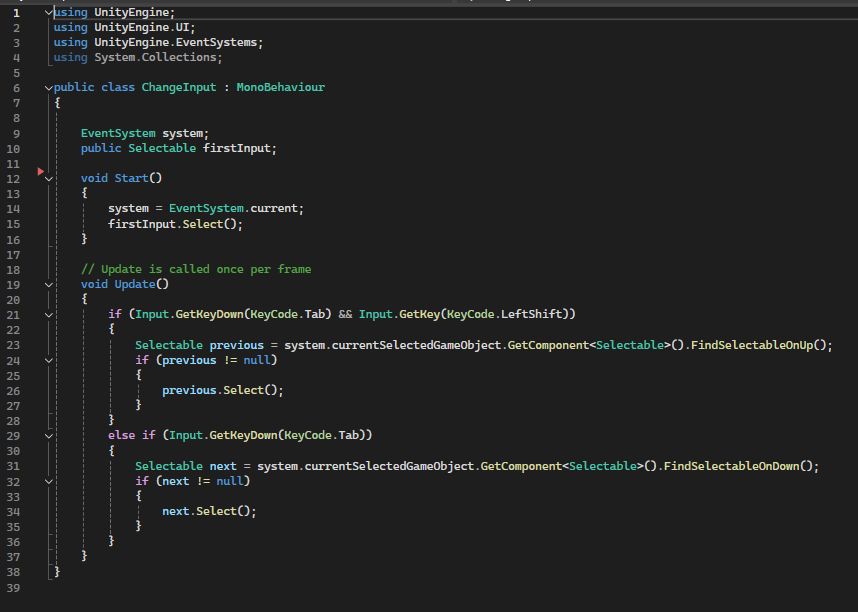
ClientServerCommunication.cs – zarządza połączeniem z serwerem, umożliwia wysyłanie i odbieranie wiadomości, oraz przetwarza odpowiedzi od serwera.

# RegisterScript.cs



Ma za zadanie zarządzać procesem rejestracji nowych użytkowników. Kiedy użytkownik wpisuje swoje dane logowania, są one pobierane i wysyłane do serwera, następnie serwer odbiera odpowiedź i na jej podstawie podejmuje zapis tych danych i przejście do następnej sceny. Jeżeli rejestracja zakończy się niepowodzeniem, użytkownik zostaje przekierowany do sceny, która informuje o nieudanej rejestracji.

# ChangeInput.cs

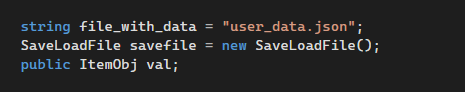


Jest to skrypt, który jest uruchomiony na scenie z logowaniem się/rejestracją. Pozwala użytkownikowi przechodzić między elementami za pomocą przycisku [TAB] w dół, lub przy wykorzystaniu kombinacji przycisków [SHIFT] + [TAB] w górę

# BuyItem.cs

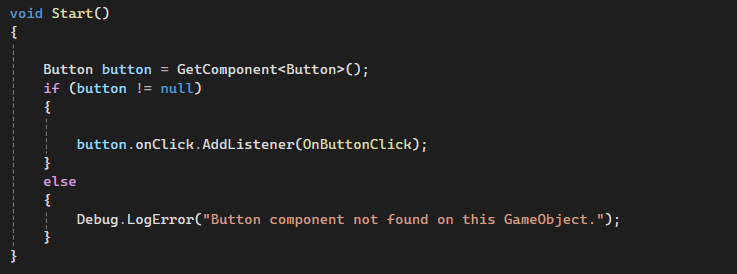
Skrypt BuyItem obsługuje interakcje związane z zakupem przedmiotów lub umiejętności przez gracza. Umożliwia aktualizację ekwipunku i umiejętności gracza oraz zapisuje te dane do pliku JSON. Wykorzystuje przyciski w interfejsie użytkownika, aby umożliwić graczowi dokonywanie zakupów.

### Pola klasy



file\_with\_data – ścieżka do pliku JSON z danymi gracza.  
savefile – instancja klasy SaveLoadFile do zapisywania i ładowania plików.  
val – publiczna zmienna typu ItemObj, przechowująca wartość przedmiotu.

### Metoda Start



Metoda Start jest wywoływana na początku. Pobiera komponent Button przypisany do tego samego obiektu. Jeśli przycisk jest znaleziony, dodaje do niego nasłuchiwacz kliknięć OnButtonClick. Jeśli przycisk nie jest znaleziony, wyświetla błąd w konsoli.

### Metoda OnButtonClick

Metoda OnButtonClick jest wywoływana po kliknięciu przycisku. Ładuje dane gracza z pliku JSON za pomocą savefile.Load\_to\_file. Parsuje załadowane dane do obiektów JObject. Sprawdza, czy val nie jest nullem. Jeśli jest, wyświetla błąd w konsoli. Pobiera tag i nazwę obiektu, a także wartość przedmiotu (itemValueInt). Sprawdza, czy gracz ma wystarczająco dużo pieniędzy na zakup przedmiotu.

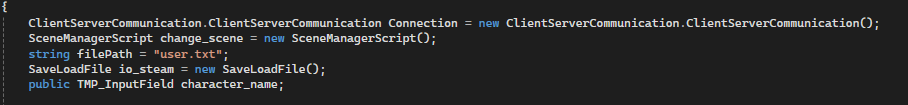
Jeśli zakup jest możliwy:

* Odejmuje koszt przedmiotu od pieniędzy gracza.
* Jeśli przedmiot nie jest umiejętnością i nie jest już w posiadaniu gracza, dodaje go do ekwipunku.
* Jeśli przedmiot jest umiejętnością i jej poziom jest mniejszy niż 5, zwiększa poziom umiejętności o 1.
* Zapisuje zaktualizowane dane do pliku JSON za pomocą savefile.Save\_to\_file.

# CreateCharacter.cs

Skrypt CreateCharacter obsługuje tworzenie nowej postaci w grze poprzez interakcję z serwerem. Wykorzystuje TMP\_InputField do pobierania nazwy postaci od użytkownika, a następnie przesyła dane do serwera. Jeśli serwer potwierdzi stworzenie nowej postaci, skrypt zmienia scenę na stronę logowania.

## Pola klasy



* Connection – instancja klasy ClientServerCommunication do obsługi komunikacji z serwerem.
* change\_scene – instancja klasy SceneManagerScript do zmiany scen.
* filePath – ścieżka do pliku z nazwą użytkownika.
* io\_steam – instancja klasy SaveLoadFile do zapisywania i ładowania plików.
* character\_name – publiczna zmienna typu TMP\_InputField przechowująca nazwę nowej postaci.

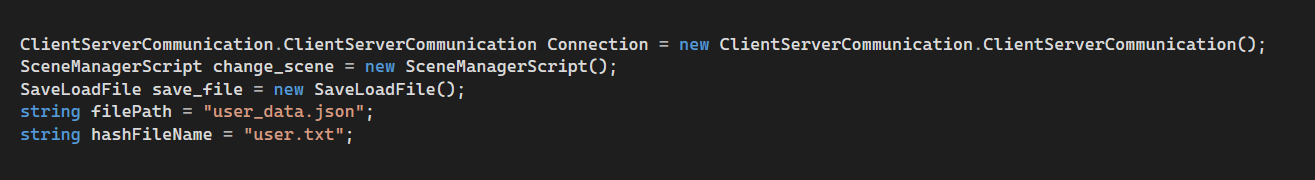
### Metoda CreateCharacter\_with\_UserLogin

Metoda CreateCharacter\_with\_UserLogin jest wywoływana do stworzenia nowej postaci. Łączy się z serwerem za pomocą Connection.ConnectToServer. Ładuje nazwę użytkownika z pliku user.txt. Tworzy JSON-owy string new\_character z danymi nowej postaci. Tworzy instancję klasy MyData z metodą PUT i wiadomością new\_character. Tworzy JSON-owy string string\_to\_add\_to\_leaderboard z danymi do dodania do tablicy wyników. Tworzy kolejną instancję klasy MyData z metodą PUT i wiadomością string\_to\_add\_to\_leaderboard. Wysyła dane nowej postaci do serwera i odbiera odpowiedź, którą loguje w konsoli. Ponownie łączy się z serwerem i wysyła dane do tablicy wyników, odbierając odpowiedź i logując ją w konsoli. Jeśli odpowiedź serwera to "add new character", zmienia scenę na "LoginPage".

# GetUserData.cs

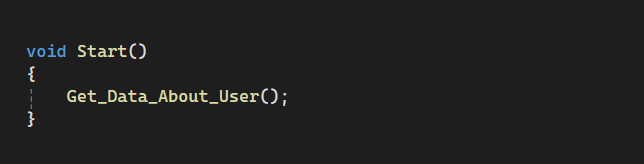
Skrypt GetUserData obsługuje pobieranie i zapisywanie danych użytkownika poprzez komunikację z serwerem. Wykorzystuje pliki tekstowe do przechowywania nazw użytkowników oraz danych użytkownika, a także wykorzystuje JSON do przesyłania danych między klientem a serwerem.

### Pola klasy



* Connection – instancja klasy ClientServerCommunication do obsługi komunikacji z serwerem.
* change\_scene – instancja klasy SceneManagerScript do zmiany scen (obecnie niewykorzystywana w tym skrypcie).
* save\_file – instancja klasy SaveLoadFile do zapisywania i ładowania plików.
* filePath – ścieżka do pliku JSON z danymi użytkownika.
* hashFileName – ścieżka do pliku z nazwą użytkownika.

### Metoda Start



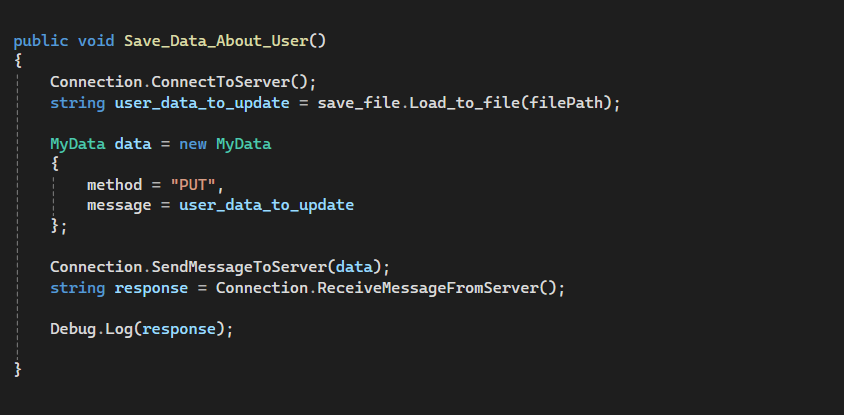
Metoda Start jest wywoływana na początku i wywołuje metodę Get\_Data\_About\_User w celu pobrania danych użytkownika.

### Metoda Get\_Data\_About\_User



Metoda Get\_Data\_About\_User łączy się z serwerem, aby pobrać dane użytkownika. Ładuje hash (nazwę użytkownika) z pliku user.txt.Tworzy JSON-owy string user\_pass\_string z danymi użytkownika. Tworzy instancję klasy MyData z metodą GET i wiadomością user\_pass\_string. Wysyła dane do serwera i odbiera odpowiedź, którą parsuje jako JSON. Jeśli odpowiedź nie jest "no player data", zapisuje pobrane dane użytkownika do pliku user\_data.json.

### Metoda Save\_Data\_About\_User

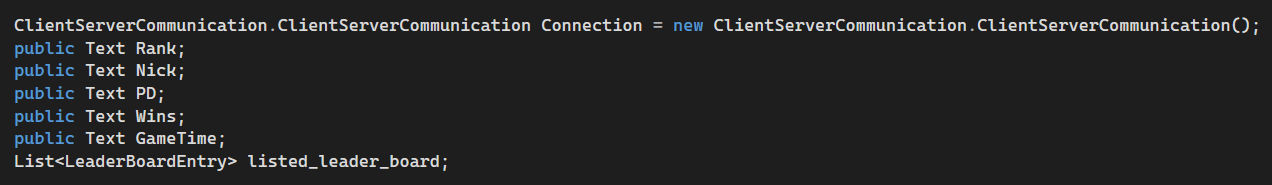


Metoda Save\_Data\_About\_User łączy się z serwerem, aby zapisać zaktualizowane dane użytkownika. Ładuje dane użytkownika z pliku user\_data.json. Tworzy instancję klasy MyData z metodą PUT i wiadomością user\_data\_to\_update. Wysyła dane do serwera i odbiera odpowiedź, którą loguje w konsoli.

# LeaderBoardLoad.cs

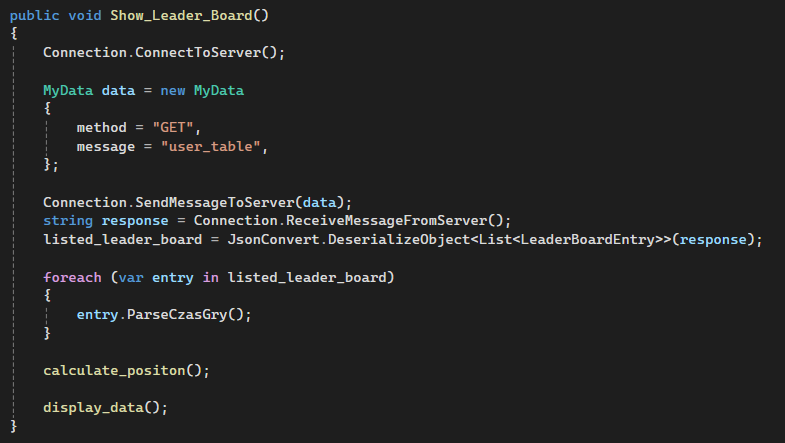
Skrypt LeaderBoardLoad obsługuje ładowanie i wyświetlanie tablicy wyników poprzez komunikację z serwerem. Pobiera dane z serwera, sortuje je na podstawie obliczonej wartości i wyświetla w odpowiednich elementach UI w Unity.

### Pola klasy



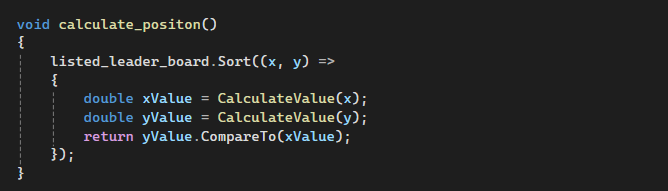
* Connection – instancja klasy ClientServerCommunication do obsługi komunikacji z serwerem.
* Rank, Nick, PD, Wins, GameTime – publiczne zmienne typu Text reprezentujące różne kolumny w tablicy wyników.
* listed\_leader\_board – lista obiektów LeaderBoardEntry reprezentująca wczytane dane z tablicy wyników.

### Metoda Show\_Leader\_Board



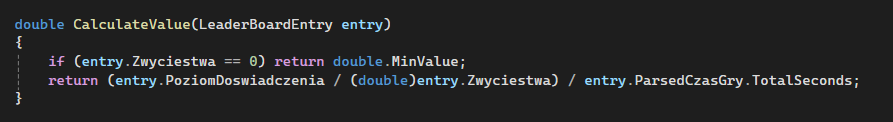
Metoda Show\_Leader\_Board jest wywoływana w celu załadowania i wyświetlenia tablicy wyników. Łączy się z serwerem za pomocą Connection. ConnectToServer.Tworzy instancję klasy MyData z metodą GET i wiadomością user\_table. Wysyła dane do serwera i odbiera odpowiedź, którą deserializuje do listy LeaderBoardEntry. Dla każdej pozycji w tablicy wyników wywołuje metodę ParseCzasGry. Wywołuje metodę calculate\_positon, aby obliczyć pozycje w tablicy wyników. Wywołuje metodę display\_data, aby wyświetlić dane.

### Metoda calculate\_positon



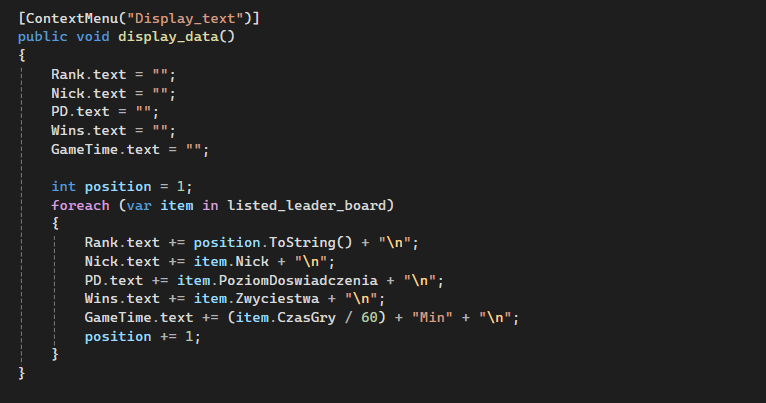
Metoda calculate\_positon sortuje tablicę wyników na podstawie wartości obliczonej przez metodę CalculateValue.

### Metoda CalculateValue



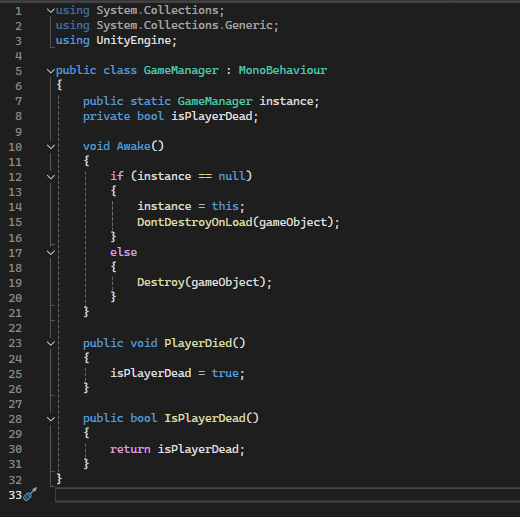
Metoda CalculateValue oblicza wartość dla każdej pozycji w tablicy wyników na podstawie poziomu doświadczenia, liczby zwycięstw i czasu gry.

### Metoda display\_data



Metoda display\_data wyświetla dane tablicy wyników w elementach UI. Czyści wcześniejsze dane w Text komponentach. Iteruje przez posortowaną tablicę wyników, przypisując odpowiednie wartości do Text komponentów.

## GameManager.cs



isPlayerDead – przechowuje informacje, czy gracz jest martwy

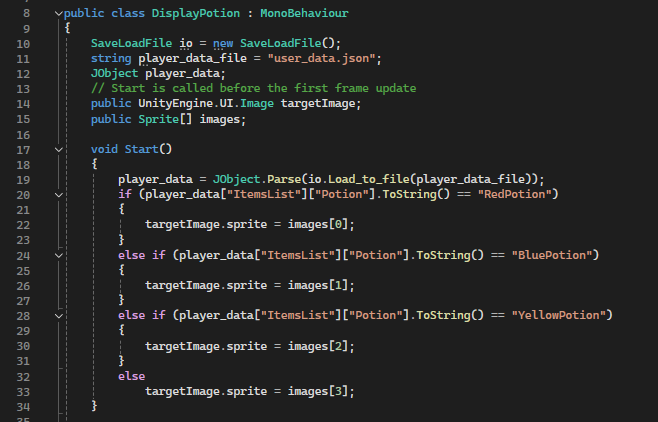
Metoda Awake – działa podobnie jak w skrypcie z muzyką, ma za zadanie nie niszczyć konkretnego obiektu, podczas zmiany sceny

Metoda PlayerDied – ustawia stan gracza na martwy

Metoda IsPlayerDead – czy gracz jest martwy

Głównym celem tej klasy jest zarządzanie stanem śmierci gracza

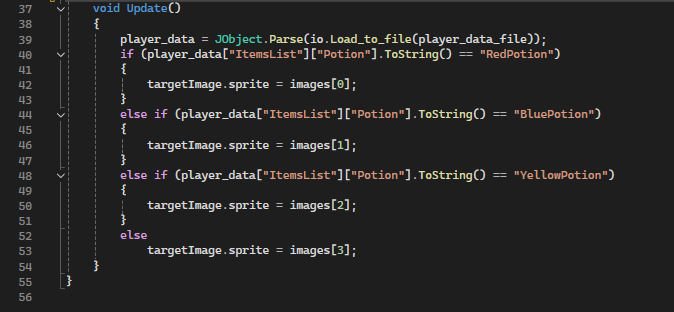
## DisplayPotion.cs



SaveLoadFile – jest odpowiedzialny za ładowanie danych z pliku, następnie wskazujemy plik z danymi gracza

Deklarujemy również zmienną, która będzie wyświetlana mikstura

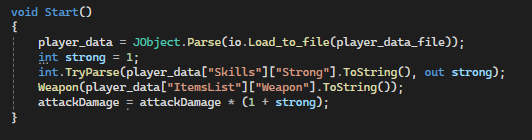
Metoda Start – ma za zadanie sprawdzić w bazie danych, jaką miksturę posiada gracz i ustawia odpowiedni obrazek



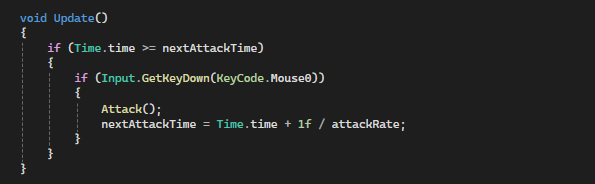
Metoda Update, ma za zadanie aktualizować obraz mikstury użytkownika w czasie rzeczywistym, jeżeli wykryje, że są zmiany w pliku danych gracza

## PlayerCombat.cs

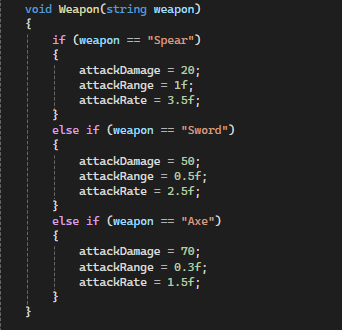
Również na samym początku deklaruje, że będzie zaczytywał dane z pliku user\_data.json, do których będzie się później skrypt odwoływał



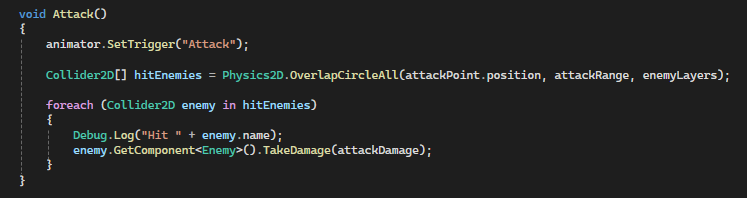
Na samym początku, zostają wczytane dane gracza i ustawiana zostaje wartość siły i broni, następnie w oparciu o siłę gracza dane są modyfikowane



Metoda update, pozwala graczowi atakować tylko wtedy, gdy minął odpowiedni czas od ostatniego ataku, jest to wykorzystywane, aby gracz nie dokonywał „spamu”, tylko wykazał się kreatywnością podczas walki



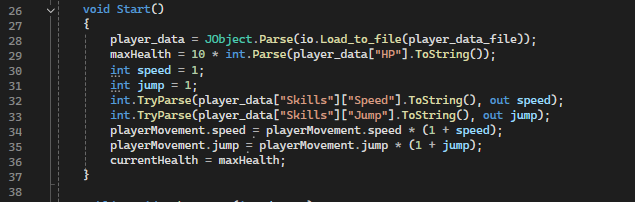
Metoda Weapon, to ustawienie konkretnych statystyk, w przypadku różnych broni



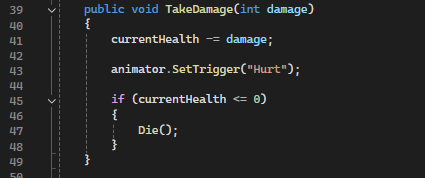
Metoda Attack, uruchamia animacje ataku, sprawdza czy przeciwnicy są w zasięgu, wypisuje w konsoli(wykorzystywaliśmy do sprawdzania, czy dobrze to działa), informacje o trafieniu i na końcu zadawane są obrażenia

## PlayerHealth.cs

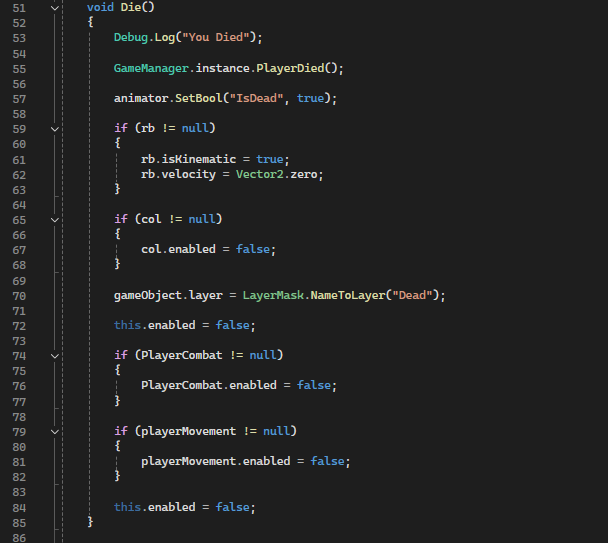
Na początku tak samo definiujemy skąd z jakiego pliku będą zaczytywane dane o graczu.



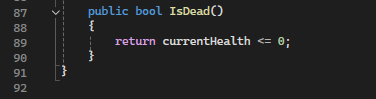
Metoda start ustawia maksymalne zdrowie, szybkość, skok gracza, a na samym końcu ustawia aktualną wartość życia na maksymalną



Metoda TakeDamage zmiejsza zdrowie gracza o zadane obrażenia i uruchamia animacje, kiedy gracz zostaje zraniony. Kiedy zdrowie gracza spadnie do zera, zostaje wywołana metoda Die



Metoda Die obsługuje proces śmierci gracza m.in. uruchamia animacje śmierci, zatrzymuje fizykę gry (żeby postać po śmierci, nadal się nie ruszała), wyłącza kolizje, zmienia warstwę obiektu na Dead. Na końcu wyłącza skrypty obsługujące motorykę gracza.

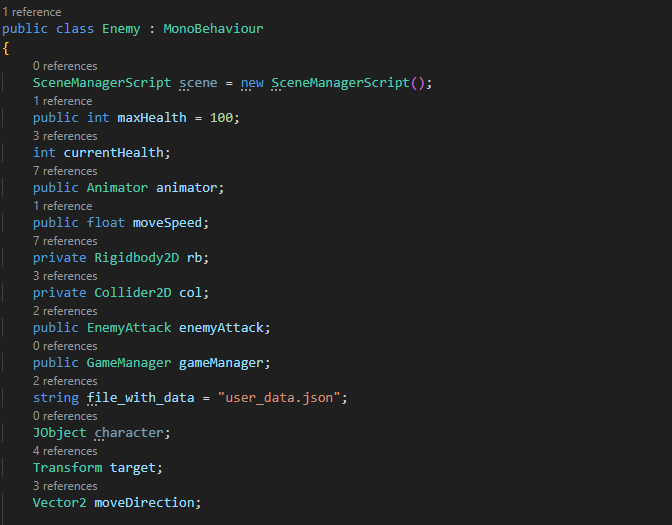


Wskazuje czy gracz jest martwy

## PlayerMovement.cs

Kod ten zarządza ruchem postaci w poziomie, skokami, kierunkiem patrzenia oraz animacjami. Wykorzystuje „Rigidbody2D” do fizyki i „Animator” do tworzenia animacji. Zarządza również kolizją.

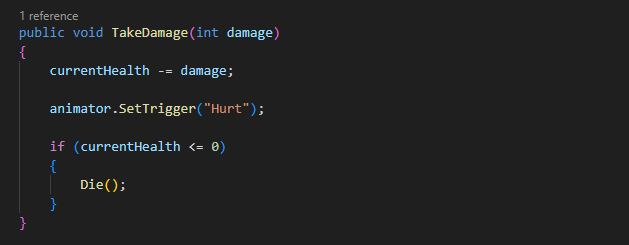
## Enemy.cs



* **SceneManagerScript scene:** Instancja skryptu SceneManagerScript, używana do zarządzania scenami w grze.
* **public int maxHealth = 100:** Maksymalne zdrowie przeciwnika ustawione na 100.
* **int currentHealth:** Bieżące zdrowie przeciwnika.
* **public Animator animator:** Animator do obsługi animacji przeciwnika.
* **public float moveSpeed:** Prędkość ruchu przeciwnika.
* **private Rigidbody2D rb:** Komponent Rigidbody2D, który pozwala na fizyczne interakcje przeciwnika.
* **private Collider2D col:** Komponent Collider2D, który pozwala na kolizje przeciwnika.
* **public EnemyAttack enemyAttack:** Skrypt odpowiedzialny za atak przeciwnika.
* **public GameManager gameManager:** Skrypt zarządzający stanem gry.
* string file\_with\_**data = "user\_data.json":** Nazwa pliku JSON zawierającego dane użytkownika.
* **JObject character:** Obiekt JSON przechowujący dane postaci.
* **Transform target:** Cel przeciwnika, którym jest gracz.
* **Vector2 moveDirection:** Kierunek ruchu przeciwnika.

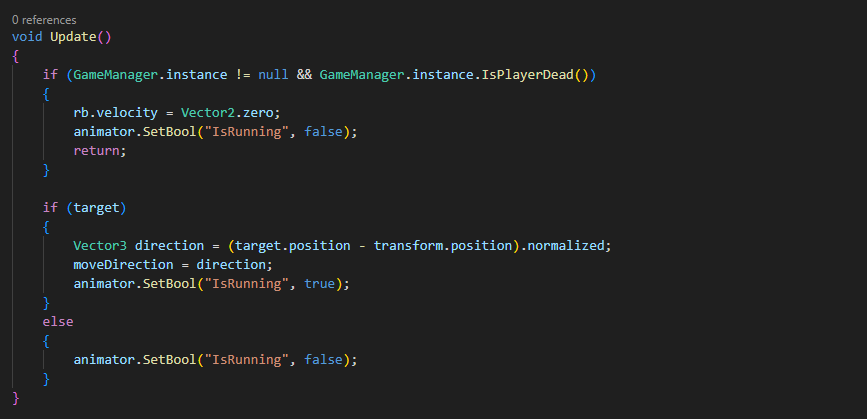


W metodzie **Start** inicjalizowane jest bieżące zdrowie przeciwnika do maksymalnej wartości, a także pobierane są komponenty Rigidbody2D i Collider2D. Skrypt wyszukuje również obiekt z tagiem "Player" i ustawia go jako cel (target).

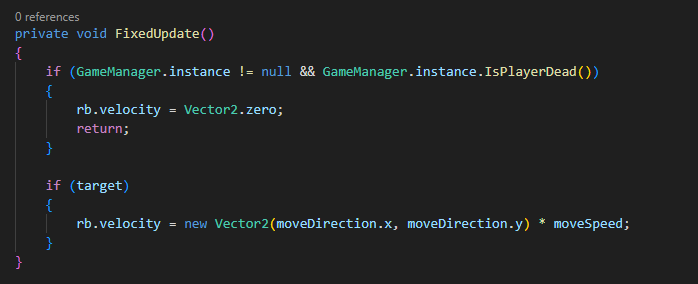


Gdy przeciwnik otrzymuje obrażenia, metoda **TakeDamage** zmniejsza jego zdrowie o podaną wartość i uruchamia animację "Hurt". Jeśli zdrowie spadnie do zera lub poniżej, wywoływana jest metoda **Die**.

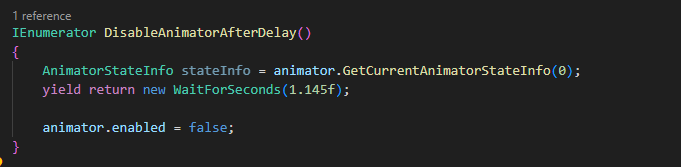
Metoda **Die** wyświetla komunikat "Enemy Died", ustawia animator na animację "IsDead" i zatrzymuje ruch przeciwnika. Komponenty fizyczne przeciwnika, takie jak Collider i Rigidbody, są dezaktywowane. Skrypt zmienia warstwę obiektu na "Dead" i wyłącza skrypt EnemyAttack. Następnie wyłącza bieżący skrypt oraz uruchamia korutynę DisableAnimatorAfterDelay, która po określonym czasie wyłącza animator. Skrypt wywołuje metodę Win w skrypcie GameManager, co prawdopodobnie oznacza wygranie gry przez gracza. Na końcu metoda Die wczytuje dane z pliku JSON, zwiększa ilość pieniędzy o 100 i zapisuje zmodyfikowane dane z powrotem do pliku.



W metodzie **Update** skrypt sprawdza, czy gracz nie jest martwy. Jeśli tak, przeciwnik przestaje się poruszać. Jeśli cel (gracz) istnieje, skrypt oblicza kierunek do celu i ustawia odpowiednie animacje ruchu.

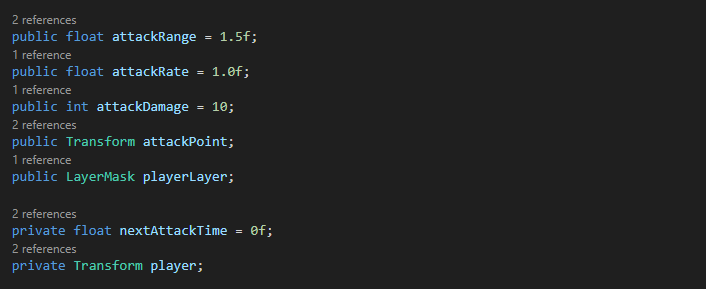


W metodzie **FixedUpdate** skrypt również sprawdza, czy gracz nie jest martwy, a jeśli cel istnieje, przeciwnik porusza się w kierunku celu z określoną prędkością.

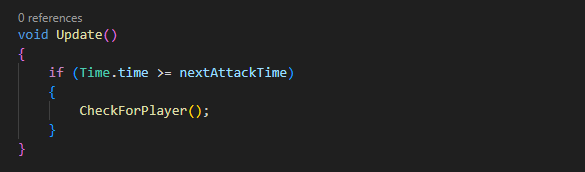


Korutyna **DisableAnimatorAfterDelay** czeka przez określony czas (1.145 sekundy) i następnie wyłącza animator, co zatrzymuje animacje przeciwnika po jego śmierci.

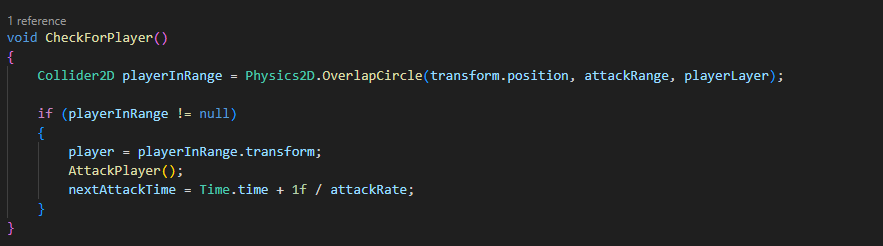
## EnemyAttack.cs



* public float attackRange = 1.5f; : Określa zasięg ataku przeciwnika w jednostkach.
* public float attackRate = 1.0f; : Określa częstotliwość ataków przeciwnika (atak na sekundę).
* public int attackDamage = 10; : Określa ilość obrażeń zadawanych przez atak przeciwnika.
* public Transform attackPoint; : Punkt, z którego przeciwnik przeprowadza atak.
* public LayerMask playerLayer; : Warstwa, do której należy gracz. Służy do identyfikacji gracza podczas ataku.
* private float nextAttackTime = 0f; : Określa czas, kiedy przeciwnik może ponownie zaatakować.
* private Transform player; : Przechowuje odniesienie do gracza, który znajduje się w zasięgu ataku.



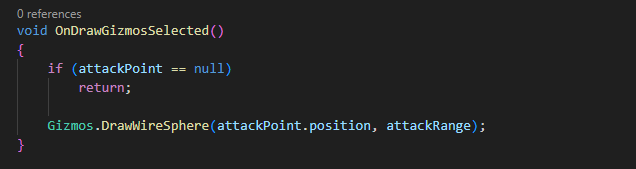
W metodzie **Update**, skrypt sprawdza, czy aktualny czas w grze jest większy lub równy czasowi nextAttackTime. Jeśli tak, wywoływana jest metoda CheckForPlayer, aby sprawdzić, czy gracz znajduje się w zasięgu ataku.



Metoda **CheckForPlayer** używa funkcji Physics2D.OverlapCircle do sprawdzenia, czy jakikolwiek obiekt na warstwie playerLayer znajduje się w promieniu attackRange od przeciwnika. Jeśli gracz znajduje się w zasięgu, jego transformacja jest przypisywana do pola player, wywoływana jest metoda AttackPlayer, a nextAttackTime jest aktualizowany na czas następnego możliwego ataku.



Metoda **AttackPlayer** uruchamia animację ataku, jeśli komponent Animator jest dostępny. Następnie zmniejsza zdrowie gracza za pomocą metody TakeDamage z komponentu PlayerHealth, o ile taki komponent istnieje na graczu.



Metoda **OnDrawGizmosSelected** rysuje w edytorze Unity wizualny zasięg ataku przeciwnika jako drucianą sferę wokół attackPoint, co pomaga w łatwiejszym ustaleniu zasięgu ataku podczas tworzenia gry.