OOP
Unity Engine
C# / .Net

Ściąga Mida

Autor
Bartłomiej Wrycza

Oto kilka popularnych zwrotów używanych w języku programowania C# i Unity, wraz z ich tłumaczeniem na język polski:

1. Variable (Zmienna) - A named storage location used to hold a value.
   Przechowuje wartość w określonym miejscu.

2. Method (Metoda) - A set of instructions or code that performs a specific task.
   Zbiór instrukcji lub kodu wykonujący określone zadanie.

3. Class (Klasa) - A blueprint or template for creating objects that define their properties and behaviors.
   Schemat lub szablon tworzenia obiektów, które definiują ich właściwości i zachowanie.

4. Object (Obiekt) - An instance of a class that represents a specific entity or concept.
   Instancja klasy reprezentująca określonego obiektu lub pojęcie.

5. Constructor (Konstruktor) - A special method used for initializing objects of a class.
   Specjalna metoda służąca do inicjalizacji obiektów danej klasy.

6. Property (Właściwość) - A member of a class that encapsulates a getter and/or setter method for accessing or modifying its value.
   Członek klasy, który enkapsuluje metody get i/lub set, umożliwiające dostęp lub modyfikację wartości.

7. Inheritance (Dziedziczenie) - The ability of a class to inherit properties and behaviors from a parent class.
   Możliwość dziedziczenia właściwości i zachowań klasy nadrzędnej przez klasę pochodną.

8. Interface (Interfejs) - A contract specifying a set of methods and properties that a class must implement.
   Kontrakt określający zestaw metod i właściwości, które klasa musi zaimplementować.

9. Event (Zdarzenie) - A mechanism for communication between objects, where an object can notify other objects when something happens.
   Mechanizm komunikacji między obiektami, w którym obiekt może powiadamiać inne obiekty o zdarzeniach.

10. Unity Editor (Edytor Unity) - A graphical interface provided by Unity for creating and editing game content.
    Graficzny interfejs dostarczany przez Unity do tworzenia i edycji zawartości gry.

11. Script (Skrypt) - A piece of code that defines the behavior or functionality of an object or a system.
    Fragment kodu definiujący zachowanie lub funkcjonalność obiektu lub systemu.

12. Debugging (Debugowanie) - The process of identifying and fixing errors or bugs in the code.
    Proces identyfikowania i naprawiania błędów lub usterek w kodzie.

Niniejszy zestaw zwrotów stanowi jedynie wstęp do tematyki programowania w języku C# i tworzenia gier w Unity. Istnieje wiele innych terminów i pojęć związanych z tymi dziedzinami.

Oto kolejne popularne zwroty związane z programowaniem .

1. Namespace (Przestrzeń nazw) - A container for organizing and grouping related classes and other types.
   Kontener służący do organizowania i grupowania powiązanych klas i innych typów.

2. Array (Tablica) - A data structure that stores a fixed-size sequential collection of elements of the same type.
   Struktura danych przechowująca kolekcję elementów tego samego typu o stałym rozmiarze.

3. Loop (Pętla) - A control structure that allows repeated execution of a block of code until a specific condition is met.
   Struktura kontrolna, która umożliwia wielokrotne wykonanie bloku kodu do momentu spełnienia określonego warunku.

4. Conditional statement (Instrukcja warunkowa) - A control structure that executes different blocks of code based on a condition.
   Struktura kontrolna, która wykonuje różne bloki kodu w zależności od spełnienia warunku.

5. Access modifier (Modyfikator dostępu) - A keyword that specifies the accessibility of a class, method, or member variable.
   Słowo kluczowe określające dostępność klasy, metody lub zmiennej składowej.

6. Return statement (Instrukcja return) - A statement that terminates the execution of a method and returns a value to the calling code.
   Instrukcja kończąca wykonanie metody i zwracająca wartość do kodu wywołującego.

7. Unity Scripting API (API skryptowe Unity) - A collection of classes and methods provided by Unity for interacting with the Unity engine.
   Zbiór klas i metod udostępnianych przez Unity do interakcji z silnikiem Unity.

8. Coroutine (Coroutine) - A special type of function that allows for asynchronous execution and timing control in Unity.
   Specjalny rodzaj funkcji, który umożliwia asynchroniczne wykonywanie i kontrolę czasową w Unity.

9. Component (Komponent) - A modular piece of functionality that can be attached to a game object in Unity to give it specific behaviors.
   Modularna funkcjonalność, która może być dołączana do obiektu gry w Unity, aby nadać mu określone zachowanie.

10. Collision (Kolizja) - An event that occurs when two objects in Unity come into contact with each other.
    Zdarzenie, które występuje, gdy dwa obiekty w Unity stykają się ze sobą.

11. Instantiate (Instancjonować) - To create a new instance of a class or object during runtime.
    Tworzenie nowej instancji klasy lub obiektu w trakcie działania programu.

12. Unity Asset Store (Sklep Assetów Unity) - An online marketplace where developers can find and download pre-made assets for use in Unity projects.
    Internetowy sklep, w którym deweloperzy mogą znajdować i pobierać gotowe assety do

wykorzystania w projektach Unity.

1. Coroutine (Coroutine) - A special type of function that allows for asynchronous execution and timing control in Unity.
Specjalny rodzaj funkcji, który umożliwia asynchroniczne wykonywanie i kontrolę czasową w Unity.

2. SerializeField (SerializeField) - An attribute used in Unity to serialize a private field, allowing it to be visible and editable in the Inspector.
Atrybut używany w Unity do serializacji prywatnego pola, umożliwiający jego wyświetlanie i edycję w Inspektorze.

3. Prefab (Prefab) - A reusable template or blueprint for creating instances of objects in Unity.
Wielokrotnie wykorzystywalny szablon lub wzorzec do tworzenia instancji obiektów w Unity.

4. GameObject (GameObject) - A fundamental object in Unity that represents a game entity or an empty container for components.
Podstawowy obiekt w Unity reprezentujący jednostkę gry lub pusty kontener na komponenty.

5. Transform (Transform) - A component attached to a GameObject that represents its position, rotation, and scale in 3D space.
Komponent dołączony do GameObjectu, który reprezentuje jego pozycję, obrót i skalę w przestrzeni 3D.

6. Rigidbody (Rigidbody) - A component used to simulate physics and enable realistic movement and collision detection for GameObjects.
Komponent używany do symulowania fizyki i umożliwiania realistycznego ruchu i detekcji kolizji dla GameObjectów.

7. OnCollisionEnter (OnCollisionEnter) - A Unity event method that is triggered when a collision occurs between two objects.
Metoda zdarzenia w Unity, która jest wywoływana, gdy występuje kolizja między dwoma obiektami.

8. Update (Update) - A Unity callback method that is called every frame and is commonly used for updating game logic.
Metoda wywoływana przez Unity w każdej klatce i często używana do aktualizacji logiki gry.

9. Coroutine (Coroutine) - A special type of function that allows for asynchronous execution and timing control in Unity.
Specjalny rodzaj funkcji, który umożliwia asynchroniczne wykonywanie i kontrolę czasową w Unity.

10. Raycasting (Raycasting) - A technique used in Unity to cast a virtual ray into the scene to detect collisions and interactions.
Technika używana w Unity do wysyłania wirtualnego promienia w scenę w celu wykrywania kolizji i interakcji.

11. Asset Bundle (Pakiet zasobów) - A collection of assets and data that can be dynamically loaded and unloaded in Unity, allowing for modular content management.
Zbiór zasobów i danych, które można dynamicznie ładować i usuwać

IEnumerable` i `IEnumerator

W języku C#, `IEnumerable` i `IEnumerator` to dwie interfejsy związane z iteracją kolekcji. Oto różnica między nimi:

1. IEnumerable:
   - `IEnumerable` jest interfejsem, który definiuje metodę `GetEnumerator()`, zwracającą obiekt implementujący interfejs `IEnumerator`.
   - `IEnumerable` reprezentuje kolekcję, która może być iterowana.
   - Dostarcza metody, takie jak `foreach`, do iteracji po elementach kolekcji.

2. IEnumerator:
   - `IEnumerator` jest interfejsem, który definiuje metody i właściwości służące do iteracji po elementach kolekcji.
   - `IEnumerator` umożliwia przechodzenie przez elementy kolekcji jeden po drugim.
   - Posiada trzy główne elementy:
     - `MoveNext()`: Przesuwa wskaźnik do następnego elementu w kolekcji i zwraca wartość logiczną wskazującą, czy jest dostępny kolejny element.
     - `Reset()`: Przesuwa wskaźnik przed pierwszy element w kolekcji.
     - `Current`: Zwraca bieżący element kolekcji.

Podsumowując, `IEnumerable` jest interfejsem, który umożliwia iterację po kolekcji, podczas gdy `IEnumerator` jest interfejsem, który zapewnia mechanizm iteracji po kolekcji, przechodząc od elementu do elementu. Kiedy chcemy iterować po kolekcji, zwykle używamy `IEnumerable`, który zwraca `IEnumerator` za pomocą metody `GetEnumerator()`.

Delegats:

Delegat to typ danych w języku programowania, który reprezentuje referencję do metody. Delegaty umożliwiają przekazywanie metod jako parametrów do innych metod, przechowywanie ich w zmiennych oraz wywoływanie ich w odpowiednim kontekście. Delegaty są używane do implementacji mechanizmu wywoływania zwrotnego (callback) oraz zdarzeń (events) w programowaniu.

Po angielsku:
A delegate is a data type in a programming language that represents a reference to a method. Delegates allow passing methods as parameters to other methods, storing them in variables, and invoking them in the appropriate context. Delegates are used to implement callback mechanisms and events in programming.

Dziedziczenie.

Dziedziczenie (ang. inheritance) to mechanizm w programowaniu obiektowym, który umożliwia tworzenie nowych klas na podstawie istniejących klas. Klasa dziedzicząca, nazywana klasą pochodną lub podrzędną, może odziedziczyć (czyli przejąć) cechy (pola i metody) z klasy bazowej, nazywanej klasą nadrzędną lub podstawową. Dziedziczenie pozwala na tworzenie hierarchii klas, gdzie klasy pochodne rozszerzają funkcjonalność klasy bazowej poprzez dodawanie nowych cech lub modyfikację już istniejących.

Przykład: Możemy mieć klasę bazową "Pojazd" z polami i metodami wspólnymi dla wszystkich pojazdów, a następnie stworzyć klasy pochodne, takie jak "Samochód" i "Motocykl", które dziedziczą (odziedziczają) cechy klasy "Pojazd" i dodają swoje unikalne cechy.

Po angielsku:
Inheritance is a mechanism in object-oriented programming that allows creating new classes based on existing classes. The derived class, also known as a subclass or child class, can inherit (i.e., acquire) the characteristics (fields and methods) from the base class, known as the superclass or parent class. Inheritance enables the creation of class hierarchies, where derived classes extend the functionality of the base class by adding new features or modifying existing ones.

Example: We can have a base class "Vehicle" with fields and methods common to all vehicles, and then create derived classes such as "Car" and "Motorcycle" that inherit the characteristics of the "Vehicle" class and add their unique features.

Programowanie asynchroniczne

Programowanie asynchroniczne (ang. asynchronous programming) to technika programowania, która umożliwia wykonywanie zadań w sposób nieblokujący, co pozwala na efektywne wykorzystanie zasobów systemu. Główną ideą programowania asynchronicznego jest uruchamianie zadań w tle, a następnie kontynuowanie wykonywania innych operacji, zamiast czekać na zakończenie każdego zadania przed przejściem do następnego.

Tradycyjne programowanie, zwane programowaniem synchronicznym, wykonuje operacje sekwencyjnie, co oznacza, że program czeka na zakończenie jednej operacji przed rozpoczęciem kolejnej. W przypadku operacji, które wymagają dużo czasu, taki sposób działania może powodować blokowanie wątku głównego, co prowadzi do spadku wydajności i reaktywności aplikacji.

Programowanie asynchroniczne wykorzystuje mechanizmy takie jak asynchroniczne metody, zadania (ang. tasks), obietnice (ang. promises) lub inne konstrukcje, które umożliwiają wykonywanie operacji w tle. Dzięki temu można kontynuować wykonywanie innych zadań lub reagować na zdarzenia, podczas gdy operacja asynchroniczna jest w trakcie realizacji. Po zakończeniu operacji asynchronicznej można przetworzyć jej wynik lub zareagować na zdarzenie z nią związane.

Programowanie asynchroniczne jest szczególnie przydatne w przypadku operacji sieciowych, dostępu do bazy danych, operacji wejścia/wyjścia, obsługi interfejsu użytkownika i innych sytuacjach, gdzie czasy odpowiedzi są długie lub niepewne. Dzięki programowaniu asynchronicznemu możliwe jest tworzenie bardziej responsywnych i wydajnych aplikacji, które lepiej wykorzystują zasoby systemu.

Oto lista metody i funkcji związanych z programowaniem asynchronicznym po polsku i angielsku:

1. Metoda asynchroniczna (async method) - Metoda, która jest oznaczona słowem kluczowym "async" i może zawierać operacje asynchroniczne.
2. Słowo kluczowe "await" - Słowo kluczowe używane wewnątrz metody asynchronicznej do oczekiwania na zakończenie operacji asynchronicznej.
3. Zadanie (Task) - Obiekt reprezentujący asynchroniczne zadanie, które może być wykonywane w tle.
4. Fabryka zadań (Task.Factory) - Klasa zawierająca metody do tworzenia zadań asynchronicznych.
5. Zdarzenie (Event) - Mechanizm używany do powiadamiania o wystąpieniu określonego zdarzenia asynchronicznego.
6. Obiekt zwracany (Return Object) - Obiekt, który jest zwracany przez metodę asynchroniczną jako wynik operacji asynchronicznej.

Po angielsku:
1. Async method - A method that is marked with the "async" keyword and can contain asynchronous operations.
2. Await keyword - A keyword used inside an async method to await the completion of an asynchronous operation.
3. Task - An object that represents an asynchronous operation that can be executed in the background.
4. Task Factory - A class that provides methods for creating asynchronous tasks.
5. Event - A mechanism used to notify about the occurrence of a specific asynchronous event.
6. Return Object - An object that is returned by an async method as the result of an asynchronous operation.

Proszę zauważyć, że terminologia związana z programowaniem asynchronicznym może nieznacznie się różnić w zależności od używanego języka programowania. Powyższe terminy odnoszą się do ogólnych koncepcji i konstrukcji występujących w programowaniu asynchronicznym.

Tak, w programowaniu asynchronicznym, obiekty `Task` są wykonywane na innym wątku niż wątek główny aplikacji.

Główny wątek aplikacji jest wątkiem, który jest odpowiedzialny za wykonywanie kodu w sekwencyjny sposób. Jednak gdy używamy asynchroniczności, możemy uruchomić operacje asynchroniczne na innych wątkach w tle, aby nie blokować głównego wątku i umożliwić równoczesne wykonywanie innych zadań.

Przy użyciu obiektów `Task` (lub `Task<T>`), możemy uruchomić zadanie asynchroniczne, które jest automatycznie przypisywane do puli wątków (thread pool) w .NET Framework. Pula wątków to grupa wątków, które są dostępne do wykonania różnych zadań asynchronicznych. Pula wątków zarządza przypisywaniem wątków do zadań w zależności od dostępności i obciążenia.

Gdy zadanie asynchroniczne jest uruchamiane, może być przypisane do jednego z wątków w puli wątków, który wykonuje to zadanie. Po zakończeniu zadania, wątek zostaje zwolniony i może być ponownie wykorzystany do innych zadań.

Dzięki temu mechanizmowi programowanie asynchroniczne umożliwia wydajne wykorzystanie zasobów systemu, a także zapewnia responsywność aplikacji, ponieważ wątek główny nie jest blokowany przez długotrwałe operacje.

Wielo Wątkowość

Semafor (Semaphore) to mechanizm synchronizacji w programowaniu wielowątkowym, który kontroluje dostęp do zasobów lub sekcji krytycznych przez wiele wątków. Semafor zapewnia możliwość ograniczenia liczby wątków, które mogą równocześnie korzystać z danego zasobu.

Oto opis kilku funkcji związanych z semaforami:

1. `Semaphore.WaitOne()`: Metoda ta blokuje bieżący wątek, jeśli semafor ma wartość 0 (brak dostępnych pozwoleń). W przeciwnym razie, jeśli semafor ma wartość większą niż 0, zmniejsza wartość semafora o 1 i pozwala wątkowi kontynuować wykonywanie.

2. `Semaphore.Release()`: Metoda ta zwiększa wartość semafora o 1, co oznacza zwolnienie jednego zezwolenia. Jeśli inne wątki czekają na semafor, zwolnienie jednego zezwolenia powoduje, że jeden z tych wątków może kontynuować swoje działanie.

3. `Semaphore(int initialCount)`: Konstruktor semafora, który przyjmuje początkową wartość semafora. Ta wartość określa liczbę dostępnych początkowo zezwoleń.

Semafor jest szczególnie przydatny w przypadkach, gdy zasób może być jednocześnie używany przez określoną liczbę wątków, na przykład przy zarządzaniu połączeniami do bazy danych, dostępem do plików lub ograniczaniu dostępu do określonych operacji.

Inne funkcje związane z semaforami to `SemaphoreSlim` - bardziej zoptymalizowana wersja semafora, `WaitHandle.WaitAny()` - oczekiwanie na dowolny semafor lub inną konkretną operację, oraz `WaitHandle.WaitAll()` - oczekiwanie na wszystkie semafory lub inne konkretne operacje. Wszystkie te funkcje służą do kontroli synchronizacji i dostępu do zasobów w programowaniu wielowątkowym.

W programowaniu wielowątkowym istnieje wiele funkcji i konstrukcji, które można używać do zarządzania i synchronizowania wielu wątków. Oto kilka przykładów:

1. Wątki (Threads): Wątki to jednostki wykonawcze, które mogą być uruchamiane równolegle i niezależnie od siebie. Można tworzyć i uruchamiać wątki, aby wykonywały różne zadania jednocześnie.

2. Blokada (Lock): Blokada to mechanizm synchronizacji, który zapewnia ekskluzywny dostęp do zasobów przez jeden wątek w danym czasie. Blokada chroni sekcje krytyczne przed dostępem równoczesnym przez wiele wątków i pomaga uniknąć problemów synchronizacyjnych.

3. Monitor: Monitor to struktura danych, która działa w połączeniu z blokadami i umożliwia wątkom synchronizację i komunikację. Można użyć monitora do wprowadzenia sekcji krytycznych i kontrolowania dostępu do współdzielonych zasobów.

4. Mutex: Mutex (Muteks) to obiekt synchronizacji, który działa podobnie do blokady, ale może być używany przez wiele procesów (nie tylko wątków) do zapewnienia ekskluzywnego dostępu do zasobów. Mutexy są szczególnie przydatne w programowaniu wielowątkowym, gdy wątki muszą synchronizować się między różnymi procesami.

5. Monitorowanie (Monitoring): Mechanizm monitorowania pozwala na monitorowanie i reagowanie na zdarzenia i zmienne w systemie wielowątkowym. Można monitorować wartości zmiennych, sygnalizować zdarzenia, synchronizować wątki i reagować na zmiany stanu.

6. Zamek (Semaphore): Semafory, o których wspomniałem wcześniej, są również używane w programowaniu wielowątkowym. Pozwalają na kontrolowanie dostępu do zasobów przez określoną liczbę wątków jednocześnie.

Te funkcje i konstrukcje służą do zarządzania wieloma wątkami, synchronizacji dostępu do zasobów, rozwiązywania problemów synchronizacyjnych i tworzenia bezpiecznych operacji wielowątkowych. Wybór konkretnego mechanizmu zależy od wymagań i charakterystyki aplikacji, jak również od preferencji programisty.

Valiut Type (Typ wartościowy):

Typ wartościowy (ang. value type) w programowaniu odnosi się do typu danych, którego wartość jest przechowywana bezpośrednio w miejscu, gdzie zmienna jest zdefiniowana. W przypadku typu wartościowego, sama wartość jest kopiowana podczas przypisania lub przekazywania do funkcji. Zmiana wartości w jednej zmiennej nie wpływa na wartość innej zmiennej.

Przykłady typów wartościowych to liczby całkowite, liczby zmiennoprzecinkowe, znaki, wartości logiczne oraz struktury (structs) zdefiniowane przez użytkownika.

Value Type:

A value type in programming refers to a data type whose value is stored directly in the location where the variable is defined. In the case of a value type, the actual value is copied when assigning or passing it to a function. Changing the value in one variable does not affect the value in another variable.

Examples of value types include integers, floating-point numbers, characters, booleans, and user-defined structures (structs).

Reference Type (Typ referencyjny):

Typ referencyjny (ang. reference type) w programowaniu odnosi się do typu danych, którego zmienna przechowuje referencję (wskaźnik) do miejsca w pamięci, gdzie jest przechowywana właściwa wartość. W przypadku typu referencyjnego, kilka zmiennych może odwoływać się do tego samego miejsca w pamięci, a zmiana wartości w jednej zmiennej wpływa na wartość innych zmiennych, które odnoszą się do tej samej referencji.

Przykłady typów referencyjnych to obiekty klas, tablice, napisy (strings) oraz wskaźniki.

W skrócie, różnica między typem wartościowym a typem referencyjnym polega na tym, jak są przechowywane i przekazywane wartości. Typy wartościowe są przechowywane jako wartości bezpośrednio w zmiennej, podczas gdy typy referencyjne przechowują referencję do miejsca w pamięci, gdzie jest przechowywana właściwa wartość.

Reference Type:

A reference type in programming refers to a data type whose variable holds a reference (pointer) to the memory location where the actual value is stored. In the case of a reference type, multiple variables can refer to the same memory location, and changing the value in one variable affects the value in other variables that refer to the same reference.

Examples of reference types include class objects, arrays, strings, and pointers.

In summary, the difference between a value type and a reference type lies in how the values are stored and passed. Value types are stored as values directly in the variable, while reference types store a reference to the memory location where the actual value is stored.

Dispose

Dispose to metoda używana w programowaniu, zwłaszcza w językach, które obsługują zarządzanie pamięcią, takich jak C# czy Java. Oznacza ona zwolnienie zasobów, takich jak pliki, strumienie, połączenia sieciowe, czy obiekty systemowe, po zakończeniu ich używania.

Kiedy korzystamy z zasobów, takich jak pliki czy połączenia sieciowe, ważne jest, aby odpowiednio nimi zarządzać i zwalniać je, gdy przestają być potrzebne. Niektóre zasoby, takie jak otwarte pliki czy połączenia sieciowe, mogą być ograniczone, a niezwolnienie ich może prowadzić do wycieków pamięci lub innych problemów z wydajnością aplikacji.

Metoda Dispose służy do zwalniania tych zasobów przed ich destrukcją. W ramach metody Dispose można zamknąć pliki, zamknąć połączenia sieciowe, zniszczyć obiekty systemowe itp. Celem Dispose jest również zapewnienie, że wszelkie zasoby, które są zarządzane przez dany obiekt, są odpowiednio zwolnione i odzyskane.

W języku C#, standardowym sposobem korzystania z metody Dispose jest użycie bloku using, który automatycznie wywołuje metodę Dispose dla obiektów, które implementują interfejs IDisposable.

Przykład użycia Dispose w C#:

```
***using (FileStream fileStream = new FileStream("file.txt", FileMode.Open))***
***{***
   ***// Wykonaj operacje na pliku***
***} // Po zakończeniu bloku using, metoda Dispose zostanie automatycznie wywołana***
```

Podsumowując, metoda Dispose służy do zwalniania zasobów po ich użyciu i powinna być wywoływana w celu zapewnienia prawidłowego zarządzania pamięcią i zasobami w programie.

Oprócz metody Dispose, istnieją także inne funkcje i mechanizmy związane z zarządzaniem i zwalnianiem zasobów. Oto kilka z nich:

1. Finalizer (Finalizer/destruktor): Finalizer jest specjalną metodą, która jest wywoływana przed zniszczeniem obiektu przez system gromadzenia śmieci (garbage collector). Służy do zwalniania zasobów, które nie zostały odpowiednio zwolnione przez metodę Dispose. Finalizer jest zaimplementowany przy użyciu specjalnego składnika języka (np. w C# jest to "~ClassName()") i nie można go bezpośrednio wywołać. Jest to mechanizm ostatniej szansy, który powinien być używany tylko wtedy, gdy nie można zapewnić prawidłowego zarządzania pamięcią i zasobami przy użyciu Dispose.

2. Using statement: Jak wspomniano wcześniej, using statement jest konstrukcją językową w C# (i innych językach), która zapewnia automatyczne wywołanie metody Dispose dla obiektów implementujących interfejs IDisposable. Using statement tworzy blok, w którym obiekt jest tworzony i używany, a po zakończeniu tego bloku metoda Dispose jest automatycznie wywoływana. Jest to wygodny sposób na zapewnienie zwalniania zasobów, nawet w przypadku wystąpienia wyjątków.

3. IDisposable interfejs: IDisposable jest interfejsem definiującym metodę Dispose. Implementacja tego interfejsu w klasie umożliwia programiście wywołanie metody Dispose na obiektach tej klasy. Jest to często używany mechanizm w języku C# do zarządzania pamięcią i zwalniania zasobów.

4. Dispose Pattern: Dispose Pattern to wzorzec projektowy, który definiuje standardowe podejście do zarządzania i zwalniania zasobów w języku C# (i innych językach). Składa się z implementacji interfejsu IDisposable, metody Dispose, finalizera (jeśli jest to konieczne) oraz obsługi flagi oznaczającej, czy obiekt został już zwolniony. Dispose Pattern zapewnia spójne i bezpieczne zarządzanie zasobami.

Te funkcje i mechanizmy są stosowane w celu zapewnienia poprawnego zarządzania pamięcią i zwalniania zasobów w programach. Dzięki nim programista ma kontrolę nad cyklem życia obiektów i może zapewnić odpowiednie zwalnianie zasobów, minimalizując wycieki pamięci i inne problemy związane z zarządzaniem zasobami.

Struct i Class

Struct i Class to dwa różne typy w językach programowania, takich jak C# czy Java, które służą do definiowania struktur danych. Oto różnice i zastosowanie obu typów:

Struct (Struktura):

- Struktury są typami wartościowymi (value types), co oznacza, że są przechowywane bezpośrednio w miejscu, gdzie są zadeklarowane.
- Struktury są zwykle mniejsze i prostsze niż klasy, ponieważ przechowują tylko dane i nie obsługują dziedziczenia, polimorfizmu ani innych zaawansowanych mechanizmów.
- Struktury są zwykle przeznaczone do reprezentowania prostych typów danych, takich jak liczby, koordynaty, daty, itp.
- Przekazanie struktury do metody odbywa się poprzez przekazanie kopii struktury, co może wpływać na wydajność w przypadku dużej struktury.
- Struktury są przekazywane jako wartość, co oznacza, że zmiana wartości w jednej instancji struktury nie wpływa na inne instancje.

Przykład zastosowania struct w C#:

```
public struct Point
{
   public int X;
   public int Y;
}

Point p = new Point();
p.X = 10;
p.Y = 20;
```

Class (Klasa):

- Klasy są typami referencyjnymi (reference types), co oznacza, że zmienna przechowuje referencję do miejsca w pamięci, gdzie obiekt klasy jest przechowywany.
- Klasy są bardziej elastyczne i zaawansowane niż struktury, ponieważ obsługują dziedziczenie, polimorfizm, enkapsulację i inne mechanizmy obiektowe.
- Klasy są przeznaczone do tworzenia bardziej skomplikowanych obiektów, które mają stan i zachowanie, oraz do organizowania kodu w hierarchii klas.
- Przekazanie klasy do metody odbywa się poprzez przekazanie referencji do obiektu, co oznacza, że zmiany w obiekcie będą widoczne w całej aplikacji.
- Klasy mogą być dziedziczone przez inne klasy, co umożliwia tworzenie hierarchii klas i tworzenie bardziej elastycznego kodu.

Przykład zastosowania class w C#:

```
***public class Person***
***{***
   ***public string Name;***
   ***public int Age;***
***}***

***Person person = new Person();***
***person.Name = "John";***
***person.Age = 30;***
```

Podsumowując, struct i class różnią się sposobem przechowywania danych i obsługiwania dziedziczenia oraz polimorfizmu. Structs są zwykle używane do reprezentowania prostych typów danych, podczas gdy classes są bardziej elastycznym narzędziem do tworzenia bardziej skomplikowanych obiektów i hierarchii klas.

Eventy

Eventy są mechanizmem w językach programowania, takich jak C# i Java, które pozwalają na tworzenie i obsługę zdarzeń w programie. Oto przykład i opis użycia eventów w obu językach:

Example of Using Events:

Let's say we have a class called "Button" that represents a graphical button in a user interface. We want to provide the ability for other parts of the program to be notified when the button is clicked. We can achieve this using events.

In C#:

```csharp
using System;

public class Button
{
    public event EventHandler Clicked; // Define the event

    public void OnClick()
    {
        if (Clicked != null)
        {
            Clicked(this, EventArgs.Empty); // Raise the event
        }
    }
}

public class Program
{
    static void Main()
    {
        Button button = new Button();
        button.Clicked += Button_Clicked; // Subscribe to the event

        // Simulate a button click
        button.OnClick();
    }

    static void Button_Clicked(object sender, EventArgs e)
    {
        Console.WriteLine("Button clicked!");
    }
}
```

Opis:
- Definiujemy klasę "Button", która ma zdarzenie "Clicked" typu "EventHandler".
- Metoda "OnClick" jest odpowiedzialna za wywołanie zdarzenia "Clicked". Sprawdzamy, czy istnieją subskrybenci zdarzenia, a następnie je wywołujemy.
- W metodzie "Main" tworzymy obiekt klasy "Button" i subskrybujemy się do zdarzenia "Clicked", podając metodę obsługującą "Button_Clicked".
- Symulujemy kliknięcie przycisku, wywołując metodę "OnClick".
- W metodzie "Button_Clicked" zostanie wyświetlony komunikat "Button clicked!".

Przykład użycia zdarzeń:

Załóżmy, że mamy klasę o nazwie "Button", która reprezentuje przycisk w interfejsie użytkownika. Chcemy umożliwić innym częściom programu otrzymywanie powiadomień, gdy przycisk zostanie kliknięty. Możemy to osiągnąć za pomocą eventów.

W języku C#:

```csharp
using System;

public class Button
{
    public event EventHandler Clicked; // Definiowanie zdarzenia

    public void OnClick()
    {
        if (Clicked != null)
        {
            Clicked(this, EventArgs.Empty); // Podnoszenie zdarzenia
        }
    }
}

public class Program
{
    static void Main()
    {
        Button button = new Button();
        button.Clicked += Button_Clicked; // Subskrypcja zdarzenia

        // Symulacja kliknięcia przycisku
        button.OnClick();
    }

    static void Button_Clicked(object sender, EventArgs e)
    {
        Console.WriteLine("Przycisk został kliknięty!");
    }
}
```

Opis:
- Definiujemy klasę "Button", która ma zdarzenie "Clicked" typu "EventHandler".
- Metoda "OnClick" jest odpowiedzialna za podniesienie zdarzenia "Clicked". Sprawdzamy, czy istnieją subskrybenci zdarzenia,

a następnie je podnosimy.
- W metodzie "Main" tworzymy obiekt klasy "Button" i subskrybujemy się do zdarzenia "Clicked", przypisując do niego metodę obsługującą "Button_Clicked".
- Symulujemy kliknięcie przycisku, wywołując metodę "OnClick".
- W metodzie "Button_Clicked" zostanie wyświetlony komunikat "Przycisk został kliknięty!".

Eventy umożliwiają tworzenie interaktywnych i reaktywnych aplikacji, które mogą informować inne części programu o wystąpieniu określonych zdarzeń. W powyższym przykładzie, gdy przycisk jest kliknięty, metoda obsługująca zdarzenie jest wywoływana i wyświetla komunikat na konsoli.

W eventach można użyć różnych funkcji, które są odpowiedzialne za obsługę zdarzeń. Oto kilka przykładów funkcji, które można wykorzystać w obsłudze eventów:

1. Void Function: Jest to standardowa funkcja, która nie zwraca żadnej wartości (typ void). Może zawierać dowolną logikę lub działania, które mają zostać wykonane w odpowiedzi na zdarzenie.

```csharp
void EventHandlerFunction(object sender, EventArgs e)
{
    // Logika obsługi zdarzenia
}
```

2. Function with Parameters: Możesz również użyć funkcji, która przyjmuje parametry, jeśli dane zdarzenia wymagają przekazania dodatkowych informacji.

```csharp
void ParameterizedFunction(object sender, CustomEventArgs e)
{
    // Logika obsługi zdarzenia z wykorzystaniem parametrów
}
```

3. Lambda Expression: Eventy mogą być obsługiwane za pomocą wyrażeń lambda, co pozwala na zwięzłą i bezpośrednią implementację kodu obsługi zdarzenia.

```csharp
button.Clicked += (sender, e) =>
{
    // Logika obsługi zdarzenia w wyrażeniu lambda
};
```

4. Anonymous Method: Można również użyć anonimowej metody jako obsługi zdarzenia. Anonimowa metoda jest bezimieniową funkcją, która jest definiowana i implementowana w miejscu, gdzie jest używana.

```csharp
button.Clicked += delegate (object sender, EventArgs e)
{
   // Logika obsługi zdarzenia w anonimowej metodzie
};
```

5. External Method: Możesz także użyć metody znajdującej się poza obszarem zdarzenia, jeśli ma ona odpowiednią sygnaturę (zgodną z delegatem zdarzenia).

```csharp
void ExternalMethod(object sender, EventArgs e)
{
   // Logika obsługi zdarzenia w zewnętrznej metodzie
}

button.Clicked += ExternalMethod; // Subskrypcja zewnętrznej metody
```

Te przykłady pokazują różne sposoby obsługi zdarzeń przy użyciu różnych funkcji. W zależności od potrzeb i kontekstu, możesz wybrać odpowiednią funkcję do obsługi danego zdarzenia.

Oto kilka funkcji związanych z obsługą eventów, które możesz użyć w języku C#:

1. `+=` (Add): Operator `+=` jest używany do subskrybowania (dodawania) funkcji obsługi zdarzenia do eventu. Przykład:
   ```csharp
   button.Clicked += Button_Clicked;
   ```

2. `-=` (Remove): Operator `-=` jest używany do odsubskrybowania (usuwanie) funkcji obsługi zdarzenia z eventu. Przykład:
   ```csharp
   button.Clicked -= Button_Clicked;
   ```

3. `Invoke`: Metoda `Invoke` służy do ręcznego wywołania zdarzenia, co jest przydatne, gdy chcesz jawnie spowodować wystąpienie zdarzenia. Przykład:
   ```csharp
   button.Clicked.Invoke(this, EventArgs.Empty);
   ```

4. `GetInvocationList`: Metoda `GetInvocationList` zwraca listę funkcji obsługi zdarzenia zapisanych w eventzie. Przykład:
```csharp
Delegate[] handlers = button.Clicked.GetInvocationList();
foreach (var handler in handlers)
{
    // Obsługa każdej funkcji obsługi zdarzenia
}
```

5. `EventName?.Invoke`: Składnia `EventName?.Invoke` jest używana do bezpiecznego wywołania zdarzenia tylko wtedy, gdy ma ono zarejestrowanych subskrybentów. Jest to przydatne, aby uniknąć wystąpienia błędu NullReferenceException, gdy zdarzenie nie ma żadnych subskrybentów. Przykład:
```csharp
button.Clicked?.Invoke(this, EventArgs.Empty);
```

6. `EventName != null`: Warunek `EventName != null` jest używany do sprawdzenia, czy zdarzenie ma jakichkolwiek subskrybentów przed jego wywołaniem. Przykład:
```csharp
if (button.Clicked != null)
{
    button.Clicked(this, EventArgs.Empty);
}
```

Te funkcje są powszechnie używane do zarządzania eventami w języku C# i pozwalają na dodawanie, usuwanie, wywoływanie i sprawdzanie subskrybentów dla danego zdarzenia.

Namespace

Namespace (przestrzeń nazw) w języku C# jest mechanizmem organizacji kodu, który pozwala na grupowanie typów i innych elementów w logiczne jednostki. Oto rozszerzenie zagadnienia namespace z przykładami zastosowania:

Namespace (Przestrzeń nazw) jest używany do:
1. Uniknięcia konfliktów nazw: Namespace umożliwia uniknięcie konfliktów nazw między różnymi elementami kodu. Można grupować powiązane klasy, struktury, interfejsy, metody itp. w odpowiednich przestrzeniach nazw, aby zapewnić unikalność nazw w kontekście danego namespace.

```csharp
namespace Przyklad1
{
    public class KlasaA { }
}

namespace Przyklad2
{
    public class KlasaA { }
}

// Użycie
Przyklad1.KlasaA obiekt1 = new Przyklad1.KlasaA();
Przyklad2.KlasaA obiekt2 = new Przyklad2.KlasaA();
```

2. Organizacji kodu: Namespace służy do logicznego organizowania kodu, zwłaszcza w większych projektach. Można tworzyć hierarchię przestrzeni nazw, aby pogrupować pokrewny kod w łatwy do zarządzania sposób.

```csharp
namespace MojaAplikacja
{
    namespace Modul1
    {
        public class KlasaA { }
        public class KlasaB { }
    }

    namespace Modul2
    {
        public class KlasaC { }
        public class KlasaD { }
    }
}
```

```
// Użycie
MojaAplikacja.Modul1.KlasaA obiekt1 = new MojaAplikacja.Modul1.KlasaA();
MojaAplikacja.Modul2.KlasaC obiekt2 = new MojaAplikacja.Modul2.KlasaC();
```

3. Ułatwienia w organizacji projektu: Namespace jest szczególnie przydatny w większych projektach, gdzie można używać różnych przestrzeni nazw dla różnych modułów lub komponentów projektu. Ułatwia to zarządzanie, nawigację i odnalezienie odpowiednich elementów w projekcie.

```csharp
namespace ProjektA
{
    public class KlasaA { }
}

namespace ProjektB
{
    public class KlasaB { }
}

// Użycie
ProjektA.KlasaA obiekt1 = new ProjektA.KlasaA();
ProjektB.KlasaB obiekt2 = new ProjektB.KlasaB();
```

4. Ułatwienia w korzystaniu z bibliotek zewnętrznych: Namespace jest używany do organizowania kodu w bibliotekach zewnętrznych. Pozwala to na łatwe rozróżnienie i używanie różnych funkcji lub klas z różnych bibliotek.

```
```

csharp
```
using BibliotekaA;
using BibliotekaB;

// Użycie
KlasaA obiekt1 = new KlasaA(); // z BibliotekaA
KlasaB obiekt2 = new KlasaB(); // z BibliotekaB
```

Namespace jest ważnym elementem w języku C#, który pomaga w organizacji kodu, unikaniu konfliktów nazw, łatwym zarządzaniu projektami i korzystaniu z bibliotek zewnętrznych. Przestrzenie nazw są używane w celu tworzenia logicznych jednostek kodu, które pomagają w zrozumieniu i utrzymaniu aplikacji.

Unity/namespace

W Unity, namespace jest również używany do organizacji kodu, podobnie jak w języku C#. Oto kilka przykładów zastosowania namespace w Unity:

1. Organizacja kodu na poziomie projektu: Można użyć namespace w Unity do organizacji kodu na poziomie projektu. Można tworzyć przestrzenie nazw dla różnych modułów, komponentów lub systemów w grze.

```csharp
namespace Game.Core
{
    public class GameManager : MonoBehaviour { }
}

namespace Game.UI
{
    public class UIManager : MonoBehaviour { }
}
```

2. Ułatwienie nawigacji i odnalezienia kodu: Namespace w Unity ułatwia nawigację po kodzie i odnalezienie odpowiednich skryptów lub komponentów w projekcie.

```csharp
using Game.Core;
using Game.UI;

public class GameController : MonoBehaviour
{
    private GameManager gameManager;
    private UIManager uiManager;

    private void Awake()
    {
        gameManager = FindObjectOfType<GameManager>();
        uiManager = FindObjectOfType<UIManager>();
    }
}
```

3. Unikanie konfliktów nazw: Namespace w Unity pozwala na uniknięcie konfliktów nazw między różnymi skryptami lub komponentami. Można grupować powiązane elementy w odpowiednich przestrzeniach nazw, aby zapewnić unikalność nazw w kontekście danego namespace.

```csharp
namespace Game.Characters
{
    public class PlayerController : MonoBehaviour { }
}

namespace Game.Items
{
    public class PlayerController : MonoBehaviour { }
}
```

4. Ułatwienie współpracy i rozszerzalności: Namespace w Unity ułatwia współpracę między członkami zespołu programistycznego, ponieważ pozwala na logiczne grupowanie skryptów i komponentów. Jest również przydatny w przypadku rozszerzania gry, ponieważ można łatwo dodawać nowe moduły lub systemy do odpowiednich przestrzeni nazw.

```csharp
namespace Game.Extensions
{
    public class PowerUpSystem : MonoBehaviour { }
}
```

W Unity, namespace służy do organizacji kodu, unikania konfliktów nazw, ułatwienia nawigacji po kodzie i współpracy w zespole. Jest to ważne narzędzie w tworzeniu skalowalnych i zorganizowanych projektów w Unity.

Interfejs

Interfejs w programowaniu jest abstrakcyjną strukturą, która definiuje zestaw metod i właściwości, które klasa implementująca ten interfejs musi zaimplementować. Interfejsy służą do definiowania kontraktu, który musi być spełniony przez klasy, które je implementują. Oto kilka kluczowych cech interfejsów:

1. Definicja metod i właściwości: Interfejsy definiują metody i właściwości, które muszą zostać zaimplementowane przez klasy. Metody w interfejsie są zadeklarowane bez implementacji, a klasy implementujące interfejs muszą dostarczyć konkretne implementacje dla tych metod.

2. Kontrakt: Interfejs definiuje kontrakt, który klasy muszą spełnić. Oznacza to, że klasy implementujące dany interfejs muszą dostarczyć wszystkie metody i właściwości zdefiniowane w interfejsie.

3. Implementacja wielu interfejsów: Klasa może implementować wiele interfejsów jednocześnie. To pozwala na implementację różnych zestawów funkcjonalności zdefiniowanych przez różne interfejsy.

4. Brak implementacji domyślnej: Metody w interfejsie nie mają domyślnej implementacji. Klasy implementujące interfejs muszą dostarczyć konkretne implementacje dla wszystkich zadeklarowanych metod.

5. Polimorfizm: Interfejsy umożliwiają polimorfizm, co oznacza, że można traktować obiekty implementujące interfejs jako instancje tego interfejsu. Dzięki temu można tworzyć kolekcje obiektów różnych klas, ale implementujących ten sam interfejs i wywoływać na nich metody zdefiniowane w interfejsie.

6. Rozszerzalność: Interfejsy pozwalają na rozszerzanie funkcjonalności klas poprzez implementację dodatkowych interfejsów. Dzięki temu można dodać nowe zestawy metod i właściwości do klasy bez zmiany jej podstawowej struktury.

Przykład interfejsu w języku C#:

```csharp
public interface ILogger
{
    void Log(string message);
    void LogError(string error);
}

public class FileLogger : ILogger
{
    public void Log(string message)
    {
        // Implementacja logiki zapisu do pliku
    }

    public void LogError(string error)
    {
        // Implementacja logiki zapisu błędów do pliku
    }
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        // Implementacja logiki wypisywania na konsolę
    }

    public void LogError(string error)
    {
        // Implementacja logiki wypisywania błędów na konsolę
    }
}
```

W powyższym przykładzie mamy interfejs `ILogger`, który definiuje dwie metody: `Log` i `LogError`. Klasy `FileLogger` i `ConsoleLogger` implementują ten interfejs,dostarczając konkretne implementacje dla tych metod. Dzięki temu można używać obiektów tych klas jako obiektów typu `ILogger` i wywoływać na nich metody zdefiniowane w interfejsie.

Interfejsy są ważnym narzędziem programowania, które umożliwiają abstrakcję, polimorfizm i rozszerzalność. Pozwalają na definiowanie wspólnego zestawu metod i właściwości, które różne klasy mogą implementować w różny sposób, jednocześnie spełniając ustalone wymagania kontraktu.

Polimorfizm i klasa abstrakcyjna

Polimorfizm i klasa abstrakcyjna to ważne koncepcje w programowaniu, w tym również w Unity. Oto ich wyjaśnienie:

1. Polimorfizm:
Polimorfizm to zdolność obiektów różnych klas do reagowania na te same metody w sposób odpowiedni do ich konkretnych implementacji. Oznacza to, że obiekty różnych klas mogą być traktowane jako instancje jednego wspólnego typu bazowego i można wywoływać na nich te same metody, jednak każda klasa może dostarczyć własną implementację tych metod. Polimorfizm umożliwia elastyczne korzystanie z różnych obiektów, które zachowują się w sposób spójny, ale różnią się w szczegółach.

W Unity, polimorfizm jest często wykorzystywany w kontekście komponentów i dziedziczenia. Na przykład, wszystkie skrypty komponentów w Unity dziedziczą po klasie `MonoBehaviour`. Dzięki temu można tworzyć różne komponenty, które implementują te same metody (np. `Start()`, `Update()`), ale dostarczają unikalną logikę dla każdego komponentu.

2. Klasa abstrakcyjna:
Klasa abstrakcyjna to klasa, która nie może być bezpośrednio instancjonowana, ale może być dziedziczona przez inne klasy. Klasa abstrakcyjna służy do definiowania ogólnych cech i zachowań, które są wspólne dla klas pochodnych. Może zawierać zarówno metody abstrakcyjne (bez implementacji), jak i metody z implementacją. Klasy dziedziczące po klasie abstrakcyjnej muszą dostarczyć implementacje wszystkich metod abstrakcyjnych zdefiniowanych w klasie bazowej.

W Unity, klasy abstrakcyjne są często używane do tworzenia podstawowych klas komponentów lub systemów, które mają pewne wspólne funkcjonalności, ale wymagają specyficznej implementacji dla każdej dziedziczącej klasy. Na przykład, można stworzyć klasę abstrakcyjną `Weapon`, która definiuje ogólne metody takie jak `Fire()`, `Reload()`, ale nie dostarcza konkretnej implementacji. Następnie można stworzyć konkretne klasy dziedziczące po klasie `Weapon`, takie jak `Pistol`, `Shotgun`, które dostarczają specyficzne implementacje tych metod.

Wniosek:
Polimorfizm i klasy abstrakcyjne są ważnymi koncepcjami w Unity, które pozwalają na elastyczne projektowanie i implementację kodu. Polimorfizm umożliwia interakcję z różnymi obiektami w spójny sposób, a klasy abstrakcyjne pozwalają na definiowanie wspólnych cech i zachowań dla klas pochodnych

. W połączeniu, te koncepcje przyczyniają się do elastycznego i skalowalnego projektowania gier w Unity.

Oto przykład klasy abstrakcyjnej w języku C#:

```csharp
public abstract class Animal
{
    public string Name { get; set; }

    public abstract void MakeSound();

    public void Eat()
    {
        Console.WriteLine("Animal is eating.");
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Woof!");
    }
}

public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meow!");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Animal dog = new Dog();
        dog.Name = "Buddy";
        dog.MakeSound(); // Output: Woof!
        dog.Eat(); // Output: Animal is eating.

        Animal cat = new Cat();
        cat.Name = "Whiskers";
        cat.MakeSound(); // Output: Meow!
        cat.Eat(); // Output: Animal is eating.
    }
}
```

W powyższym przykładzie mamy klasę abstrakcyjną `Animal`, która definiuje właściwość `Name` i abstrakcyjną metodę `MakeSound()`. Klasa `Animal` zawiera również niemutowalną metodę `Eat()`, która ma implementację.

Klasy `Dog` i `Cat` dziedziczą po klasie `Animal` i implementują jej abstrakcyjną metodę `MakeSound()`. Każda z tych klas dostarcza własnej implementacji metody `MakeSound()`.

W klasie `Program` tworzymy instancje obiektów `Dog` i `Cat` jako obiekty typu `Animal`. Możemy wywoływać metody `MakeSound()` i `Eat()` na tych obiektach, korzystając z polimorfizmu. Ostateczne wyjście programu będzie różne, zależnie od konkretnych implementacji w klasach `Dog` i `Cat`.

Oto przykład deklaracji pustej metody abstrakcyjnej w klasie abstrakcyjnej:

```csharp
public abstract class Shape
{
    public abstract void Draw();

    public void Resize(int newSize)
    {
        // Logika zmiany rozmiaru kształtu
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}

public class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a rectangle");
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Shape circle = new Circle();
        circle.Draw(); // Output: Drawing a circle
        circle.Resize(10);

        Shape rectangle = new Rectangle();
        rectangle.Draw(); // Output: Drawing a rectangle
        rectangle.Resize(20);
    }
}
```

W powyższym przykładzie mamy klasę abstrakcyjną `Shape`, która zawiera deklarację pustej metody abstrakcyjnej `Draw()`. Metoda `Draw()` jest odpowiedzialna za rysowanie konkretnego kształtu, ale nie ma implementacji w klasie bazowej `Shape`.

Klasy `Circle` i `Rectangle` dziedziczą po klasie `Shape` i dostarczają swoje konkretne implementacje metody `Draw()`. Każda z tych klas musi dostarczyć własną implementację metody `Draw()`, w zgodności z wymaganiami abstrakcyjnej metody w klasie bazowej.

W klasie `Program` tworzymy instancje obiektów `Circle` i `Rectangle` jako obiekty typu `Shape`. Możemy wywoływać metodę `Draw()` na tych obiektach, korzystając z polimorfizmu. Ostateczne wyjście programu będzie zależeć od konkretnych implementacji w klasach `Circle` i `Rectangle`.

Typy generyczne

Typy generyczne (generic types) to mechanizm w językach programowania, który umożliwia tworzenie uniwersalnych, parametryzowanych typów, które mogą być używane dla różnych rodzajów danych. Pozwalają one na tworzenie bardziej ogólnych i elastycznych rozwiązań, które są niezależne od konkretnych typów danych.

W języku C# i Unity, typy generyczne są szeroko wykorzystywane, szczególnie w kolekcjach, algorytmach i interfejsach. Oto kilka przykładów zastosowania typów generycznych:

1. Kolekcje generyczne:
   C# dostarcza wiele kolekcji generycznych, takich jak `List<T>`, `Dictionary<TKey, TValue>`, `Queue<T>`, itp. Pozwalają one na przechowywanie i manipulację danymi różnych typów w sposób bezpieczny i efektywny.

   Przykład:
   ```csharp
   List<string> names = new List<string>();
   names.Add("John");
   names.Add("Jane");

   Dictionary<int, string> ages = new Dictionary<int, string>();
   ages.Add(25, "John");
   ages.Add(30, "Jane");
   ```
2. Metody generyczne:
   Metody generyczne umożliwiają definiowanie ogólnych operacji, które mogą być wykorzystywane dla różnych typów danych. Dzięki nim można uniknąć powielania kodu i napisać bardziej elastyczne funkcje.

   Przykład:
   ```csharp
   public T GetMax<T>(T a, T b) where T : IComparable<T>
   {
      return a.CompareTo(b) > 0 ? a : b;
   }

   int maxInt = GetMax(5, 10);
   float maxFloat = GetMax(3.14f, 2.5f);
   ```

3. Interfejsy generyczne:
Interfejsy generyczne pozwalają na tworzenie ogólnych kontraktów, które mogą być implementowane przez różne typy danych. Pozwalają one na zapewnienie wspólnego zestawu funkcji i operacji dla różnych typów, bez wymuszania konkretnej implementacji.

Przykład:
```csharp
public interface IRepository<T>
{
    void Add(T item);
    void Remove(T item);
    T GetById(int id);
}

public class UserRepository : IRepository<User>
{
    public void Add(User user)
    {
        // Implementacja dodawania użytkownika
    }

    public void Remove(User user)
    {
        // Implementacja usuwania użytkownika
    }

    public User GetById(int id)
    {
        // Implementacja pobierania użytkownika po ID
    }
}
```

Typy generyczne są potężnym narzędziem, które umożliwiają tworzenie bardziej ogólnych i elastycznych rozwiązań programistycznych. Pozwalają na ponowne użycie kodu, zwiększają bezpieczeństwo typów i ułatwiają tworzenie bardziej generycznych bibliotek i frameworków.

Oto kilka często wykorzystywanych bibliotek w Unity:

1. UnityEngine:
Jest to główna biblioteka Unity, która zawiera wiele kluczowych klas i funkcji do tworzenia gier. Zawiera m.in. klasy dotyczące transformacji obiektów, renderowania grafiki, obsługi fizyki, dźwięku, kolizji, wejścia, animacji i wiele innych.

2. System.Collections:
Biblioteka System.Collections zawiera różne kolekcje generyczne i nongeneryczne, takie jak List, Dictionary, Queue, Stack, itp. Są one używane do przechowywania i manipulacji danymi w grze.

3. System.IO:
Biblioteka System.IO dostarcza funkcje do operacji wejścia/wyjścia, takie jak odczyt i zapis plików, tworzenie folderów, operacje na strumieniach danych, itp.

4. System.Math:
Biblioteka System.Math zawiera wiele funkcji matematycznych, takich jak obliczanie pierwiastków, potęg, funkcji trygonometrycznych, logarytmów, zaokrąglanie liczb, itp.

5. UnityEngine.UI:
Biblioteka UnityEngine.UI zawiera komponenty do tworzenia interfejsu użytkownika w Unity, takie jak przyciski, teksty, obrazy, suwaki, panele, itp. Pozwala na tworzenie interaktywnych i responsywnych UI w grze.

6. System.Diagnostics:
Biblioteka System.Diagnostics zawiera funkcje do profilowania i debugowania aplikacji. Można jej używać do pomiaru czasu wykonywania operacji, śledzenia wydajności, tworzenia logów, itp.

7. UnityEngine.Networking:
Biblioteka UnityEngine.Networking dostarcza narzędzia do tworzenia sieciowych funkcji w grach, takich jak komunikacja klient-serwer, synchronizacja stanu gry, obsługa protokołów sieciowych, itp.

Te biblioteki są tylko kilkoma z wielu dostępnych w Unity. Każda z nich posiada wiele funkcji i klas, które wspierają różne aspekty tworzenia gier i aplikacji.

System.Collections

Oto krótkie opisy działania wybranych kolekcji:

1. List<T>:
- List<T> jest dynamiczną listą elementów o określonym typie T.
- Można dodawać, usuwać, indeksować i sortować elementy w liście.
- List<T> przechowuje elementy w kolejności dodawania.
- Zapewnia dynamiczne rozszerzanie i zmniejszanie rozmiaru wewnętrznej tablicy w celu obsługi zmian liczby elementów.

2. ArrayList:
- ArrayList jest dynamiczną listą obiektów.
- Może przechowywać elementy różnych typów, ponieważ traktuje je jako obiekty.
- Elementy są przechowywane w kolejności dodawania.
- Zapewnia dynamiczne rozszerzanie i zmniejszanie rozmiaru wewnętrznej tablicy w celu obsługi zmian liczby elementów.

3. Queue<T>:
- Queue<T> to kolejka FIFO (First-In-First-Out) dla elementów typu T.
- Nowe elementy są dodawane na końcu kolejki, a usuwane są z początku kolejki.
- Można dodawać, usuwać i sprawdzać elementy w kolejce.

4. Stack<T>:
- Stack<T> to stos LIFO (Last-In-First-Out) dla elementów typu T.
- Nowe elementy są dodawane na wierzch stosu, a usuwane są również z wierzchu stosu.
- Można dodawać, usuwać i sprawdzać elementy na stosie.

5. LinkedList<T>:
   - LinkedList<T> to dwukierunkowa lista, w której każdy element przechowuje referencje do poprzedniego i następnego elementu.
   - Można dodawać, usuwać i przeglądać elementy w liście w dowolnej kolejności.
   - Zapewnia efektywne operacje wstawiania i usuwania elementów w dowolnym miejscu na liście.

6. HashSet<T>:
   - HashSet<T> to kolekcja unikalnych elementów typu T.
   - Zapewnia efektywną wyszukiwanie elementów bez konieczności zachowania określonej kolejności.
   - Nie pozwala na duplikaty elementów.

7. SortedSet<T>:
   - SortedSet<T> to kolekcja unikalnych elementów typu T, które są przechowywane w uporządkowanej kolejności.
   - Elementy są automatycznie sortowane przy dodawaniu do zbioru.
   - Zapewnia efektywne wyszukiwanie elementów oraz operacje na przedziałach elementów.

8. Dictionary<TKey, TValue>:
   - Dictionary<TKey, TValue> to kolekcja par klucz-wartość, gdzie klucze są unikalne.
   - Można dodawać, usuwać, pobierać i aktualizować wartości na podstawie kluczy.
   - Zapewnia efektywne wyszukiwanie wartości na podstawie klucza.

To tylko krótki opis działania wybranych kolekcji w Unity. Każda z tych kolekcji ma swoje specyficzne cechy i metody, które można wykorzystać do manipulacji danymi.

Oto dodatkowo funkcje sortujące dostępne w niektórych kolekcjach:

1. List<T>:
   - Sort(): Sortuje elementy w liście zgodnie z domyślnym porządkiem sortowania.
   - Sort(IComparer<T> comparer): Sortuje elementy w liście przy użyciu określonego komparatora.
   - Sort(Comparison<T> comparison): Sortuje elementy w liście przy użyciu określonego delegata porównującego.

2. ArrayList:
   - Sort(): Sortuje elementy w ArrayList zgodnie z domyślnym porządkiem sortowania.
   - Sort(IComparer comparer): Sortuje elementy w ArrayList przy użyciu określonego obiektu porównującego.

3. Array:
   - Sort(Array array): Sortuje elementy w tablicy zgodnie z domyślnym porządkiem sortowania.
   - Sort(Array array, IComparer comparer): Sortuje elementy w tablicy przy użyciu określonego obiektu porównującego.

W przypadku innych kolekcji, takich jak Queue<T>, Stack<T>, LinkedList<T>, HashSet<T>, SortedSet<T>, Dictionary<TKey, TValue>, SortedDictionary<TKey, TValue>, ObservableCollection<T>, nie ma bezpośrednio dostępnych funkcji sortowania. W przypadku tych kolekcji, jeśli chcesz posortować ich elementy, musisz najpierw skonwertować je na inną kolekcję, która obsługuje sortowanie, np. List<T>, a następnie użyć funkcji sortujących dostępnych dla List<T>.

Oto lista różnych funkcji i metod, które można używać w różnych kolekcjach w bibliotece System.Collections:

1. List<T>:
   - Add(T item): Dodaje element na koniec listy.
   - Remove(T item): Usuwa pierwsze wystąpienie określonego elementu z listy.
   - RemoveAt(int index): Usuwa element z określonego indeksu.
   - Clear(): Usuwa wszystkie elementy z listy.
   - Contains(T item): Sprawdza, czy lista zawiera określony element.
   - Count: Zwraca liczbę elementów w liście.
   - IndexOf(T item): Znajduje indeks pierwszego wystąpienia określonego elementu w liście.
   - Insert(int index, T item): Wstawia element na określoną pozycję.

2. ArrayList:
   - Add(object value): Dodaje element na koniec listy.
   - Remove(object value): Usuwa pierwsze wystąpienie określonego elementu z listy.
   - RemoveAt(int index): Usuwa element z określonego indeksu.
   - Clear(): Usuwa wszystkie elementy z listy.
   - Contains(object value): Sprawdza, czy lista zawiera określony element.
   - Count: Zwraca liczbę elementów w liście.
   - IndexOf(object value): Znajduje indeks pierwszego wystąpienia określonego elementu w liście.
   - Insert(int index, object value): Wstawia element na określoną pozycję.

3. Queue<T>:
   - Enqueue(T item): Dodaje element do kolejki.
   - Dequeue(): Usuwa i zwraca pierwszy element z kolejki.
   - Peek(): Zwraca pierwszy element w kolejce, nie usuwając go.
   - Clear(): Usuwa wszystkie elementy z kolejki.
   - Contains(T item): Sprawdza, czy kolejka zawiera określony element.
   - Count: Zwraca liczbę elementów w kolejce.

4. Stack<T>:
   - Push(T item): Dodaje element na szczyt stosu.
   - Pop(): Usuwa i zwraca element ze szczytu stosu.
   - Peek(): Zwraca element na szczycie stosu, nie usuwając go.
   - Clear(): Usuwa wszystkie elementy ze stosu.
   - Contains(T item): Sprawdza, czy stos zawiera określony element.
   - Count: Zwraca liczbę elementów na stosie.

5. LinkedList<T>:
   - AddFirst(T value): Dodaje element na początek listy.
   - AddLast(T value): Dodaje element na koniec listy.
   - Remove(T value): Usuwa pierwsze wystąpienie określonego elementu z listy.
   - RemoveFirst(): Usuwa pierwszy element z listy.
   - RemoveLast(): Usuwa ostatni element z listy.
   - Clear(): Usuwa wszystkie elementy z listy.
   - Contains(T value): Sprawdza, czy lista zawiera określony element.
   - Count: Zwraca liczbę elementów w liście.

6. HashSet<T>:
   - Add(T item): Dodaje element do zbioru.
   - Remove(T item): Usuwa element ze zbioru.
   - Clear(): Usuwa wszystkie elementy ze zbioru.
   - Contains(T item): Sprawdza, czy zbiór zawiera określony element.
   - Count: Zwraca liczbę elementów w zbiorze.

7. SortedSet<T>:
   - Add(T item): Dodaje element do posortowanego zbioru.
   - Remove(T item): Usuwa element ze zbioru.
   - Clear(): Usuwa wszystkie elementy ze zbioru.
   - Contains(T item): Sprawdza, czy zbiór zawiera określony element.
   - Count: Zwraca liczbę elementów w zbiorze.

8. Dictionary<TKey, TValue>:
   - Add(TKey key, TValue value): Dodaje parę klucz-wartość do słownika.
   - Remove(TKey key): Usuwa parę klucz-wartość o określonym kluczu ze słownika.
   - Clear(): Usuwa wszystkie pary klucz-wartość ze słownika.
   - ContainsKey(TKey key): Sprawdza, czy słownik zawiera określony klucz.
   - ContainsValue(TValue value): Sprawdza, czy słownik zawiera określoną wartość.
   - Count: Zwraca liczbę par klucz-wartość w słowniku.

9. SortedDictionary<TKey, TValue>:
   - Add(TKey key, TValue value): Dodaje parę klucz-wartość do posortowanego słownika.
   - Remove(TKey key): Usuwa parę klucz-wartość o określonym kluczu ze słownika.
   - Clear(): Usuwa wszystkie pary klucz-wartość ze słownika.
   - ContainsKey(TKey key): Sprawdza, czy słownik zawiera określony klucz.
   - ContainsValue(TValue value): Sprawdza, czy słownik zawiera określoną wartość.
   - Count: Zwraca liczbę par klucz-wartość w słowniku.

10. ObservableCollection<T>:
    - Add(T item): Dodaje element do kolekcji powiadamiając o zmianach.
    - Remove(T item): Usuwa element z kolekcji powiadamiając o zmianach.
    - Clear(): Usuwa wszystkie elementy z kolekcji powiadamiając o zmianach.
    - Contains(T item): Sprawdza, czy kolekcja zawiera określony element.
    - Count: Zwraca liczbę elementów w kolekcji.

To tylko wybrane funkcje i metody dostępne w poszczególnych kolekcjach. Każda z kolekcji posiada również wiele innych funkcji i metod, które można wykorzystać w zależności od potrzeb.

Oto przykłady deklaracji różnych kolekcji w języku C#:

1. List<T>:
   ```csharp
   List<int> numbers = new List<int>();
   ```

2. ArrayList:
   ```csharp
   ArrayList arrayList = new ArrayList();
   ```

3. Queue<T>:
```csharp
Queue<string> queue = new Queue<string>();
```

4. Stack<T>:
```csharp
Stack<double> stack = new Stack<double>();
```

5. LinkedList<T>:
```csharp
LinkedList<char> linkedList = new LinkedList<char>();
```

6. HashSet<T>:
```csharp
HashSet<string> hashSet = new HashSet<string>();
```

7. SortedSet<T>:
```csharp
SortedSet<int> sortedSet = new SortedSet<int>();
```

8. Dictionary<TKey, TValue>:
```csharp
Dictionary<string, int> dictionary = new Dictionary<string, int>();
```

9. SortedDictionary<TKey, TValue>:
```csharp
SortedDictionary<string, float> sortedDictionary = new SortedDictionary<string, float>();
```

10. ObservableCollection<T>:
```csharp
ObservableCollection<Person> collection = new ObservableCollection<Person>();
```

W powyższych przykładach `T` oznacza typ danych, który zostanie przechowywany w danej kolekcji, a `numbers`, `arrayList`, `queue`, itd. to nazwy zmiennych, które można dostosować według potrzeb.

Oto przykłady odwoływania się do poszczególnych wartości w różnych kolekcjach:

1. List<T>:
```csharp
List<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };
int firstNumber = numbers[0];
int lastNumber = numbers[numbers.Count - 1];
```

2. ArrayList:
```csharp
ArrayList arrayList = new ArrayList() { "apple", "banana", "cherry" };
string fruit = (string)arrayList[1];
```

3. Queue<T>:
```csharp
Queue<string> queue = new Queue<string>();
queue.Enqueue("first");
queue.Enqueue("second");
string firstElement = queue.Peek();
string dequeuedElement = queue.Dequeue();
```

4. Stack<T>:
```csharp
Stack<double> stack = new Stack<double>();
stack.Push(3.14);
stack.Push(2.718);
double topElement = stack.Peek();
double poppedElement = stack.Pop();
```

5. LinkedList<T>:
```csharp
LinkedList<char> linkedList = new LinkedList<char>();
linkedList.AddLast('A');
linkedList.AddLast('B');
char firstElement = linkedList.First.Value;
char lastElement = linkedList.Last.Value;
```

6. HashSet<T>:
```csharp
HashSet<string> hashSet = new HashSet<string>() { "apple", "banana", "cherry" };
bool containsBanana = hashSet.Contains("banana");
```

7. SortedSet<T>:
```csharp
SortedSet<int> sortedSet = new SortedSet<int>() { 3, 1, 2, 5, 4 };
int firstElement = sortedSet.First();
int lastElement = sortedSet.Last();
```

8. Dictionary<TKey, TValue>:
```csharp
Dictionary<string, int> dictionary = new Dictionary<string, int>();
dictionary.Add("one", 1);
dictionary.Add("two", 2);
int value = dictionary["one"];
```

W powyższych przykładach używam różnych metod i właściwości dostępnych w każdej kolekcji do odwołania się do poszczególnych wartości. Pamiętaj, że odwołanie się do wartości w kolekcji może różnić się w zależności od typu kolekcji i dostępnych metod.

Oto kilka przykładów funkcji i metod dostępnych w bibliotece `System.IO`:

1. File:
   - File.Exists(string path): Sprawdza, czy plik o podanej ścieżce istnieje.
   - File.Open(string path, FileMode mode): Otwiera istniejący plik w określonym trybie (do odczytu, zapisu, itp.).
   - File.Create(string path): Tworzy nowy plik o podanej ścieżce.
   - File.Copy(string sourceFileName, string destFileName): Kopiuje plik z jednej lokalizacji do drugiej.
   - File.Delete(string path): Usuwa określony plik.
   - File.ReadAllText(string path): Odczytuje zawartość pliku jako pojedynczy ciąg znaków.
   - File.WriteAllBytes(string path, byte[] bytes): Zapisuje tablicę bajtów do pliku.

2. Directory:
   - Directory.CreateDirectory(string path): Tworzy nowy katalog o podanej ścieżce.
   - Directory.Exists(string path): Sprawdza, czy katalog o podanej ścieżce istnieje.
   - Directory.GetFiles(string path): Zwraca tablicę ścieżek do plików w określonym katalogu.
   - Directory.GetDirectories(string path): Zwraca tablicę ścieżek do katalogów w określonym katalogu.
   - Directory.Move(string sourceDirName, string destDirName): Przenosi katalog z jednej lokalizacji do drugiej.
   - Directory.Delete(string path): Usuwa określony katalog i jego zawartość rekursywnie.

3. Path:
   - Path.Combine(params string[] paths): Łączy podane ścieżki w jedną ścieżkę.
   - Path.GetFullPath(string path): Zwraca pełną ścieżkę dla określonej ścieżki względnej.
   - Path.GetDirectoryName(string path): Zwraca nazwę katalogu dla określonej ścieżki.
   - Path.GetTempFileName(): Zwraca unikalną nazwę tymczasowego pliku.

4. StreamReader:
   - StreamReader(string path): Tworzy nową instancję `StreamReader` dla odczytu danych z pliku.
   - ReadLine(): Odczytuje następną linię z pliku.
   - Read(): Odczytuje kolejny znak z pliku.
   - Peek(): Zwraca następny dostępny znak bez przechodzenia do następnej linii.
   - ReadToEnd(): Odczytuje całą zawartość pliku jako pojedynczy ciąg znaków.

5. StreamWriter:
   - StreamWriter(string path): Tworzy nową instancję `StreamWriter` dla zapisu danych do pliku.
   - WriteLine(string value): Zapisuje podaną wartość do pliku, zakończoną nową linią.
   - Write(string value): Zapisuje podaną wartość do pliku.
   - Flush(): Zapisuje wszelkie buforowane dane na dysk.
   - Close(): Zamyka strumień i zwalnia wszystkie zasoby związane z nim.

To tylko niektóre z

funkcji i metod dostępnych w bibliotece `System.IO`. Istnieje wiele innych, które umożliwiają bardziej zaawansowane operacje na plikach i katalogach, takie jak kopiowanie, przenoszenie, usuwanie, itp. Zachęcam do zapoznania się z dokumentacją biblioteki `System.IO` w celu uzyskania pełnej listy funkcji i metod oraz szczegółowych informacji o ich użyciu.

Oto kilka przykładów funkcji i metod dostępnych w bibliotece `System.Math`:

- Math.Abs(double value): Zwraca wartość bezwzględną podanej liczby.
- Math.Sqrt(double value): Oblicza pierwiastek kwadratowy z podanej liczby.
- Math.Round(double value): Zaokrągla podaną liczbę do najbliższej liczby całkowitej.
- Math.Max(double x, double y): Zwraca większą wartość spośród dwóch podanych liczb.
- Math.Min(double x, double y): Zwraca mniejszą wartość spośród dwóch podanych liczb.
- Math.Pow(double x, double y): Podnosi podaną liczbę do określonej potęgi.
- Math.Sin(double angle): Oblicza sinus kąta podanego w radianach.
- Math.Cos(double angle): Oblicza cosinus kąta podanego w radianach.
- Math.Tan(double angle): Oblicza tangens kąta podanego w radianach.
- Math.PI: Stała reprezentująca wartość liczby π (pi).

To tylko kilka przykładów funkcji i stałych dostępnych w bibliotece `System.Math`. Biblioteka ta zawiera wiele innych funkcji matematycznych, które umożliwiają wykonywanie różnych operacji matematycznych, takich jak logarytmy, funkcje trygonometryczne, zaawansowane zaokrąglanie, itp. Zachęcam do dalszego eksplorowania dokumentacji biblioteki `System.Math` w celu poznania pełnej listy funkcji i metod oraz szczegółowych informacji o ich użyciu.

UnityEngine.Networking

Biblioteka `UnityEngine.Networking` jest częścią silnika Unity i zapewnia narzędzia i funkcje do tworzenia gier sieciowych i aplikacji opartych na sieci w Unity. Oto kilka przykładów funkcji i klas dostępnych w `UnityEngine.Networking`:

1. NetworkManager: Klasa odpowiedzialna za zarządzanie siecią w grze. Udostępnia funkcje do tworzenia, dołączania i zarządzania sesjami sieciowymi.

2. NetworkIdentity: Klasa reprezentująca identyfikator sieciowy obiektu w grze. Umożliwia synchronizację danych między klientami i serwerem.

3. NetworkTransform: Komponent, który umożliwia synchronizację pozycji, rotacji i skali obiektu między klientami i serwerem.

4. NetworkBehaviour: Klasa bazowa dla komponentów, które mają być używane w grach sieciowych. Zapewnia funkcje do przesyłania danych między klientami i serwerem.

5. NetworkConnection: Klasa reprezentująca połączenie sieciowe między klientem a serwerem. Umożliwia wysyłanie i odbieranie danych między stronami.

6. NetworkMessage: Klasa bazowa dla wiadomości sieciowych, które można przesyłać między klientami i serwerem.

7. NetworkTransport: Klasa do niskopoziomowej komunikacji sieciowej. Zapewnia funkcje do przesyłania danych poprzez protokoły takie jak TCP i UDP.

To tylko niektóre z klas i funkcji dostępnych w bibliotece `UnityEngine.Networking`. Ta biblioteka zapewnia szereg narzędzi i funkcji do tworzenia i zarządzania siecią w grach Unity. Zachęcam do zapoznania się z dokumentacją Unity lub odwiedzenia ich strony internetowej w celu uzyskania pełnej listy funkcji i metod oraz szczegółowych informacji o ich użyciu w kontekście Unity Networking.

UnityEngine

Oto połączona lista przykładowych funkcji dostępnych w bibliotece `UnityEngine`:

1. Debug:
   - Debug.Log(string message): Wyświetla wiadomość w konsoli deweloperskiej.
   - Debug.LogWarning(string message): Wyświetla ostrzeżenie w konsoli deweloperskiej.
   - Debug.LogError(string message): Wyświetla błąd w konsoli deweloperskiej.

2. Input:
   - Input.GetKey(KeyCode key): Sprawdza, czy dany klawisz jest wciśnięty.
   - Input.GetMouseButtonDown(int button): Sprawdza, czy określony przycisk myszy został kliknięty.
   - Input.GetAxis(string axisName): Pobiera wartość osi ruchu lub osi sterowania.
   - Input.GetButtonDown(string buttonName): Sprawdza, czy przycisk na kontrolerze został wciśnięty.

3. Time:
   - Time.deltaTime: Zwraca czas, jaki upłynął od ostatniej klatki.
   - Time.timeScale: Kontroluje prędkość czasu w grze.

4. Transform:
   - Transform.position: Pozycja obiektu w przestrzeni.
   - Transform.rotation: Rotacja obiektu.
   - Transform.localScale: Skala obiektu.

5. GameObject:
   - GameObject.Find(string name): Znajduje obiekt o określonej nazwie w scenie.
   - GameObject.SetActive(bool value): Ustawia widoczność obiektu na podstawie wartości logicznej.

6. Rigidbody:
   - Rigidbody.AddForce(Vector3 force): Dodaje siłę do komponentu Rigidbody.
   - Rigidbody.velocity: Prędkość obiektu fizycznego.

7. Collider:
   - Collider.enabled: Włącza/wyłącza kolizję na komponencie Collider.
   - Collider.OnTriggerEnter(Collider other): Metoda wywoływana przy wejściu w strefę wyzwalania.

8. Physics:
   - Physics.Raycast(Vector3 origin, Vector3 direction, float maxDistance): Wykonuje promień (raycast) i zwraca informacje o trafieniu.
   - Physics.SphereCast(Vector3 origin, float radius, Vector3 direction, float maxDistance): Wykonuje promień sferyczny (spherecast) i zwraca informacje o trafieniu.

9. SceneManager:
   - SceneManager.LoadScene(string sceneName): Ładuje nową scenę na podstawie jej nazwy.
   - SceneManager.LoadSceneAsync(string sceneName): Asynchronicznie ładuje nową scenę na podstawie jej nazwy.

10. AudioSource:
    - AudioSource.Play(): Rozpoczyna odtwarzanie dźwięku.
    - AudioSource.Stop(): Zatrzymuje odtwarzanie dźwięku.

To tylko niektóre przykłady funkcji dostępnych w `UnityEngine`. Ta biblioteka oferuje wiele narzędzi i funkcji do tworzenia gier i aplikacji w Unity. Zachęcam do zgłębienia dokumentacji Unity, aby uzyskać pełną listę funkcji i metod w `UnityEngine` oraz szczegółowe informacje na temat ich użycia.

Oto wypisane funkcje i metody z klasy `MonoBehaviour` w kolejności ich wykonywania podczas cyklu życia obiektu w Unity:

1. Awake(): Wywoływana raz, tuż po utworzeniu obiektu. Wykorzystywana do inicjalizacji obiektu przed rozpoczęciem jakiejkolwiek innej logiki.

2. OnEnable(): Wywoływana, gdy obiekt zostaje włączony lub aktywowany. Może być używana do przywracania stanu obiektu po wyłączeniu lub inicjalizacji po włączeniu.

3. Start(): Wywoływana raz, tuż przed rozpoczęciem pierwszej klatki. Wykorzystywana do inicjalizacji logiki obiektu, takiej jak znalezienie referencji do innych obiektów.

4. FixedUpdate(): Wywoływana z określoną częstotliwością w zależności od ustawień fizyki. Wykorzystywana do logiki związanej z fizyką obiektu.

5. Update(): Wywoływana raz na każdą klatkę. Wykorzystywana do ogólnej logiki obiektu.

6. LateUpdate(): Wywoływana raz na każdą klatkę, po zakończeniu wywołań funkcji Update(). Wykorzystywana do logiki, która musi być wykonana po zaktualizowaniu pozostałych obiektów w scenie.

7. OnDisable(): Wywoływana, gdy obiekt zostaje wyłączony lub dezaktywowany. Wykorzystywana do czyszczenia zasobów lub zatrzymywania działań obiektu.

8. OnDestroy(): Wywoływana, gdy obiekt jest niszczony. Wykorzystywana do zwalniania zasobów i wykonywania czynności przed zniszczeniem obiektu.

Powyższa kolejność wywoływania funkcji i metod dotyczy komponentów `MonoBehaviour` podłączonych do obiektów w scenie Unity. Jest to ogólny przewodnik po cyklu życia obiektu, ale w zależności od potrzeb i implementacji, nie zawsze wszystkie funkcje muszą być używane. Ważne jest dostosowanie ich do konkretnej logiki i wymagań projektu.

Ray

Ray (promień) jest narzędziem wykorzystywanym w grafice komputerowej i symulacji fizyki do reprezentowania ścieżki światła lub trajektorii obiektu w trójwymiarowej przestrzeni. W kontekście gier i aplikacji interaktywnych w Unity, Ray jest szeroko stosowany do detekcji kolizji, interakcji z obiektami oraz implementacji systemów strzałów, promieniowania czy ukierunkowanego ruchu.

Zastosowanie Ray w Unity polega na emisji promienia z określonego punktu w przestrzeni w określonym kierunku. Można następnie sprawdzić, czy promień przecina się z innymi obiektami, takimi jak kolizje fizyczne, elementy interaktywne lub inne elementy świata gry. Pozwala to na wykrywanie trafień, zbieranie informacji o trafionych obiektach oraz reagowanie na te zdarzenia.

Przykład zastosowania Ray w Unity może obejmować:
1. Strzelanie: Ray jest używany do wystrzelenia promienia z broni gracza w celu sprawdzenia, czy trafiono w przeciwnika, obiekt interaktywny lub inny element świata gry.
2. Wykrywanie kolizji: Ray jest używany do wykrywania kolizji między obiektami. Można użyć go do sprawdzenia, czy gracz koliduje z innymi obiektami lub czy dwa obiekty na scenie się przecinają.
3. Ukierunkowany ruch: Ray może być używany do określenia kierunku, w którym obiekt powinien poruszać się w przestrzeni. Na podstawie trafienia promienia w inny obiekt, można dostosować ruch lub rotację obiektu.

W Unity, do emisji promienia i sprawdzania kolizji używana jest funkcja `Physics.Raycast`. Przyjmuje ona jako argumenty pozycję początkową promienia, kierunek promienia oraz maksymalną odległość, jaką promień może przebyć. Funkcja zwraca informacje o trafieniu, takie jak trafiony obiekt, odległość od początku promienia itp.

Przykład użycia Ray w Unity może wyglądać następująco:

```csharp
RaycastHit hit;
if (Physics.Raycast(transform.position, transform.forward, out hit, maxDistance))
{
   // Promień trafił w obiekt
   GameObject hitObject = hit.collider.gameObject;
   float distance = hit.distance;
   // Wykonaj akcję w zależności od trafienia
   // np. obrażenie przeciwnika, aktywacja elementu interaktywnego itp.
}
```

W tym przykładzie promień jest wystrzeliwany z pozycji obiektu i w kierunku, w którym jest zwrócony. Jeśli promień trafi w jakiś ob

iekt na swojej drodze, informacje o trafieniu są przechwytywane, a odpowiednie działania są podejmowane w zależności od trafienia.

Ray jest potężnym narzędziem w Unity, które umożliwia interakcję obiektów w przestrzeni gry. Może być używany do rozwiązywania różnorodnych problemów, takich jak wykrywanie kolizji, strzelanie, ukierunkowany ruch czy nawet tworzenie zaawansowanych systemów optymalizacji.

Layers i Tag

W Unity, "layers" i "tagi" są mechanizmami używanymi do kategoryzowania i identyfikowania obiektów w scenie gry. Pozwalają one na bardziej precyzyjne zarządzanie i manipulację obiektami, oraz ułatwiają interakcję między nimi.

Layers (warstwy):
- Warstwy pozwalają na logiczne grupowanie obiektów w scenie gry.
- Każdy obiekt w Unity może być przypisany do jednej z dostępnych warstw.
- Pozwala to na łatwiejsze manipulowanie obiektami i kontrolowanie ich zachowania.
- Przykładowe warstwy mogą obejmować warstwę postaci gracza, przeciwników, tła itp.
- Layers są często używane w kontekście detekcji kolizji, renderingu czy systemów interakcji.

Tagi:
- Tagi są używane do nadawania identyfikatorów obiektom w celu oznaczenia ich specyficzną kategorią lub rolą.
- Pozwalają na łatwiejsze wyszukiwanie, identyfikację i manipulację obiektami na podstawie ich przypisanych tagów.
- Każdy obiekt może mieć jeden tag, choć tagi mogą być współdzielone przez wiele obiektów.
- Przykładowe tagi mogą obejmować tag gracza, przeciwnika, skarbów, pułapek itp.
- Tagi są często używane w systemach interakcji, skryptach zarządzających lub do separacji specjalnych obiektów w scenie.

Przykłady użycia:
1. Detekcja kolizji: Można ustawić warstwy i tagi dla obiektów w celu określenia, które obiekty powinny być uwzględniane przy detekcji kolizji. Na przykład, obiekt z warstwą "Player" może wykrywać kolizje z obiektami z warstwy "Enemy".
2. Manipulacja obiektami: Tagi mogą być używane do łatwiejszego odnajdywania i manipulacji obiektami. Na przykład, skrypt zarządzający grą może wyszukiwać obiekty o określonym tagu "Treasure" w celu zwiększenia wyniku gracza.
3. Renderowanie: Można używać warstw do kontrolowania renderingu obiektów. Na przykład, obiekty na warstwie "Background" mogą być renderowane jako tło, podczas gdy obiekty na warstwie "Foreground" mogą być renderowane jako przedni plan.

Layers i tagi są przydatnymi mechanizmami w Unity, które pomagają w organizacji sceny gry, detekcji kolizji, zarządzaniu obiektami i wielu innych zastosowaniach.

Oto przykłady wykrywania kolizji z wykorzystaniem promieni (Ray) i tagów w Unity:

1. Wykrywanie kolizji z użyciem Ray i tagu:

```csharp
RaycastHit hit;
if (Physics.Raycast(transform.position, transform.forward, out hit))
{
    if (hit.collider.CompareTag("Enemy"))
    {
        // Promień trafił na obiekt o tagu "Enemy"
        GameObject enemy = hit.collider.gameObject;
        // Wykonaj akcję odpowiednią dla trafienia na przeciwnika
    }
    else if (hit.collider.CompareTag("Pickup"))
    {
        // Promień trafił na obiekt o tagu "Pickup"
        GameObject pickup = hit.collider.gameObject;
        // Wykonaj akcję odpowiednią dla trafienia na zbieralny obiekt
    }
    // Dodaj więcej warunków dla innych tagów, jeśli jest to potrzebne
}
```

W powyższym przykładzie promień jest wystrzeliwany z pozycji obiektu i w określonym kierunku (`transform.forward`). Jeśli promień trafi na jakiś obiekt, sprawdzamy tag tego obiektu za pomocą `hit.collider.CompareTag()`. Na podstawie tagu trafionego obiektu wykonujemy odpowiednie działania.

2. Wykrywanie kolizji z użyciem Ray i warstw:

```csharp
RaycastHit hit;
int enemyLayer = LayerMask.GetMask("Enemy");
int pickupLayer = LayerMask.GetMask("Pickup");

if (Physics.Raycast(transform.position, transform.forward, out hit))
{
    if (hit.collider.gameObject.layer == enemyLayer)
    {
        // Promień trafił na obiekt na warstwie "Enemy"
        GameObject enemy = hit.collider.gameObject;
        // Wykonaj akcję odpowiednią dla trafienia na przeciwnika
    }
    else if (hit.collider.gameObject.layer == pickupLayer)
    {
        // Promień trafił na obiekt na warstwie "Pickup"
        GameObject pickup = hit.collider.gameObject;
        // Wykonaj akcję odpowiednią dla trafienia na zbieralny obiekt
    }
    // Dodaj więcej warunków dla innych warstw, jeśli jest to potrzebne
}
```

W tym przykładzie używamy funkcji `LayerMask.GetMask()` do pobrania wartości warstwy na podstawie jej nazwy. Następnie, porównujemy warstwę trafionego obiektu z wartościami warstw za pomocą `hit.collider.gameObject.layer`. Na podstawie warstwy trafionego obiektu wykonujemy odpowiednie działania.

W obu przypadkach możesz dostosować warunki i działania do swoich potrzeb, dodając odpowiednie tagi lub warstwy oraz określając akcje, które mają być wykonane po trafieniu na obiekt z danym tagiem lub na określoną warstwę.

Oto przykłady zapisu zmiennej jako tag lub warstwa w Unity:

1. Zapisanie zmiennej jako tag:
```csharp
string enemyTag = "Enemy";
gameObject.tag = enemyTag;
```
W powyższym przykładzie zmienna `enemyTag` przechowuje wartość "Enemy", a następnie przypisujemy tę wartość jako tag naszego obiektu `gameObject`.

2. Zapisanie zmiennej jako warstwa:
```csharp
int enemyLayer = LayerMask.NameToLayer("Enemy");
gameObject.layer = enemyLayer;
```
W tym przykładzie używamy funkcji `LayerMask.NameToLayer()` do przypisania zmiennej `enemyLayer` wartości odpowiedniej warstwy na podstawie jej nazwy "Enemy". Następnie przypisujemy tę wartość jako warstwę naszego obiektu `gameObject`.

Pamiętaj, że podczas przypisywania tagów lub warstw należy upewnić się, że używane nazwy tagów lub warstw są zgodne z zdefiniowanymi w Unity. W przeciwnym razie przypisanie nie zostanie poprawnie wykonane.

Zapisanie zmiennej jako tag lub warstwa pozwala na dynamiczne ustalanie identyfikacji obiektów w trakcie działania gry. Możesz używać zmiennych do przechowywania wartości tagów lub warstw i przypisywać je obiektom w zależności od potrzeb.

Const

W języku programowania, `const` to skrót od słowa "constant" (stała). Jest to specjalne słowo kluczowe, które służy do deklarowania wartości, które nie mogą być modyfikowane po ich przypisaniu. Wartości `const` są traktowane jako wartości stałe i są znane na etapie kompilacji.

Oto kilka cech i zastosowań `const`:

1. Stała wartość: Deklaracja zmiennej jako `const` oznacza, że wartość tej zmiennej nie może być zmieniana po jej przypisaniu. Jest to przydatne w przypadku wartości, które powinny pozostać niezmienne i znane na etapie kompilacji.

2. Kompilacja: Wartości `const` są znane na etapie kompilacji, co oznacza, że ich wartości są wprowadzane bezpośrednio do kodu wynikowego. W przeciwieństwie do zmiennych, które mogą mieć wartość przypisaną w trakcie działania programu, wartości `const` są ustalane podczas kompilacji i nie mogą być zmieniane w trakcie działania programu.

3. Wykorzystanie w obliczeniach: `const` może być używane do definiowania stałych matematycznych, takich jak wartość Pi, współczynniki fizyczne itp. Wartości te mogą być używane w obliczeniach bez konieczności ponownego przeliczania ich wartości za każdym razem.

4. Optymalizacja: Użycie `const` może pomóc kompilatorowi w optymalizacji kodu. Ponieważ wartości `const` są znane na etapie kompilacji, kompilator może dokonać pewnych optymalizacji, takich jak stałe zastępowanie wartości w kodzie, co może prowadzić do bardziej efektywnego kodu wynikowego.

Przykład użycia `const`:
```csharp
const float Gravity = 9.8f;
const int MaxScore = 100;
```
W tym przykładzie zdefiniowano dwie stałe: `Gravity` o wartości 9.8 i `MaxScore` o wartości 100. Te wartości są stałe i nie mogą być zmieniane w trakcie działania programu.

Stosowanie `const` jest przydatne, gdy mamy wartości, które powinny być stałe i znane na etapie kompilacji. Umożliwia to bardziej wyraźne i bezpieczne programowanie, eliminując możliwość przypadkowej modyfikacji wartości.

Struct

Struktura (struct) to typ wartościowy w języku programowania, który umożliwia grupowanie różnych zmiennych o różnych typach danych w jedną jednostkę. Struktury są podobne do klas, ale mają pewne różnice w zachowaniu i zastosowaniu.

Oto kilka przydatnych zastosowań struktur:

1. Przechowywanie danych: Struktury mogą być wykorzystywane do przechowywania i organizowania danych o różnych typach w jednym obiekcie. Na przykład, możemy stworzyć strukturę "Point" do przechowywania współrzędnych x, y i z:

```csharp
public struct Point
{
   public float x;
   public float y;
   public float z;
}
```

2. Efektywna pamięć: Struktury są przekazywane jako wartości, co oznacza, że są kopiowane w całości, a nie przekazywane przez referencję jak w przypadku klas. Dzięki temu struktury są bardziej efektywne pod względem zużycia pamięci i czasu wykonania w przypadku przechowywania małych ilości danych.

3. Implementacja interfejsów: Struktury mogą implementować interfejsy, co umożliwia im definiowanie kontraktów i korzystanie z polimorfizmu. Może to być przydatne, gdy chcemy, aby nasza struktura działała z różnymi typami interfejsów.

4. Przechowywanie danych bezstanowych: Struktury są szczególnie przydatne, gdy chcemy przechowywać dane bezstanowe, które nie mają metod i nie przechowują stanu. Mogą one służyć jako proste kontenery na dane, które nie wymagają bardziej zaawansowanych funkcjonalności klas.

5. Operacje na danych: Struktury mogą zawierać metody, które umożliwiają manipulację danymi wewnątrz struktury. Na przykład, możemy dodać metodę do struktury "Point", która oblicza odległość od innego punktu:

```csharp
public struct Point
{
   public float x;
   public float y;
   public float z;

   public float DistanceTo(Point other)
   {
      float dx = other.x - x;
      float dy = other.y - y;
      float dz = other.z - z;
      return Mathf.Sqrt(dx * dx + dy * dy + dz * dz);
   }
}
```

Struktury są szczególnie przydatne, gdy chcemy przechowywać i manipulować danymi w bardziej lekkim i bezstanowym sposobie. Jednakże, należy pamiętać, że struktury mają inne zasady działania niż klasy, a ich używanie zależy od konkretnego przypadku i wymagań projektu.

Używanie struktur (struct) w programowaniu ma swoje zalety, wady i pewne potencjalne niebezpieczeństwa. Oto kilka aspektów, które warto wziąć pod uwagę:

Zalety:

1. Efektywność pamięciowa: Struktury są przekazywane jako wartości, co oznacza, że są kopiowane w całości, a nie przekazywane przez referencję. To może prowadzić do efektywniejszego wykorzystania pamięci, zwłaszcza gdy przechowujemy małe ilości danych.

2. Bezstanowość: Struktury są często używane do przechowywania danych bezstanowych, które nie mają stanu ani zależności od innych obiektów. Działa to dobrze w przypadku prostych kontenerów na dane.

3. Wydajność: Dzięki ich charakterystyce wartościowej i mniejszej złożoności, struktury mogą być bardziej wydajne niż klasy w niektórych przypadkach, szczególnie gdy operujemy na dużej ilości danych.

Wady:

1. Kopia wartości: Przekazywanie struktury jako wartości może prowadzić do tworzenia kopii, co może zwiększać zużycie pamięci i wpływać na wydajność. W przypadku większych struktur lub operacji na nich może być bardziej efektywne przekazywanie ich przez referencję za pomocą `ref` lub `in`.

2. Brak dziedziczenia: Struktury nie mogą dziedziczyć po innych strukturach ani po klasach. Mają one ograniczone możliwości rozszerzania i reużywalności w porównaniu do klas.

3. Brak wskaźników null: Struktury nie mogą przyjmować wartości null, ponieważ są typami wartościowymi. Może to wprowadzać pewne ograniczenia w przypadku potrzeby reprezentowania braku wartości.

Niebezpieczeństwa:

1. Pomyłki w stosowaniu: Niewłaściwe użycie struktur, takie jak przekazywanie ich przez referencję lub mutowanie pól struktury, może prowadzić do nieprzewidywalnego zachowania. Wskazane jest zrozumienie zasad i ograniczeń związanych z korzystaniem ze struktur.

2. Niejednoznaczność: Struktury o takiej samej strukturze pól mogą mieć różne wyniki operacji porównania (np. `==`). To wynika z tego, że są porównywane na podstawie wartości pól, a nie referencji.

3. Skomplikowane operacje: Struktury, zwłaszcza te bardziej skomplikowane, mogą wymagać dodatkowej uwagi przy przekazywaniu, klonowaniu czy innych operacjach, aby uniknąć nieoczekiwanych wyników.

Podsumowując, struktury są przydatne w przypadkach przechowywania małych, bezstanowych danych i mogą zapewnić wydajność i efektywność pamięciową. Jednakże, należy zachować ostrożność i zrozumienie ich zasad działania, aby uniknąć potencjalnych pułapek i błędów.

Chociaż struktury (struct) mają swoje zastosowania i korzyści, w niektórych przypadkach klasy (class) mogą być lepszym wyborem. Oto kilka powodów, dla których klasa może być preferowana nad strukturą:

1. Referencyjna semantyka: Klasy są przekazywane przez referencję, co oznacza, że przekazywane jest tylko odwołanie do obiektu, a nie cały obiekt. Dzięki temu zmiany dokonywane na obiekcie w jednym miejscu będą widoczne w innych miejscach, co ułatwia zarządzanie stanem i unikanie niepotrzebnych kopii danych.

2. Dziedziczenie i rozszerzalność: Klasy mogą dziedziczyć po innych klasach, co pozwala na tworzenie hierarchii klas i ponowne wykorzystywanie kodu. Dzięki dziedziczeniu można rozszerzać funkcjonalność klasy bazowej, co jest przydatne w bardziej złożonych scenariuszach.

3. Wskaźniki null: Klasy mogą przyjmować wartość null, co oznacza brak przypisanego obiektu. To daje większą elastyczność w obsłudze przypadków, w których obiekt może nie być dostępny lub może wystąpić sytuacja, w której brak obiektu jest ważną informacją.

4. Zmienność: Klasy mogą zawierać zarówno pola, jak i metody, które mogą być modyfikowalne. Dzięki temu można dynamicznie zmieniać stan i zachowanie obiektu w trakcie działania programu.

5. Bardziej zaawansowane funkcje: Klasy mają więcej zaawansowanych funkcji i możliwości niż struktury, takie jak destruktory, zdarzenia, operatory przeciążenia, interfejsy, metody rozszerzające itp.

6. Rozmiar i kopiowanie danych: Struktury, zwłaszcza te większe, są kopiowane w całości, co może prowadzić do większego zużycia pamięci i wpływać na wydajność. Klasy, jako typy referencyjne, są przekazywane przez referencję, co zmniejsza narzut związany z kopiowaniem danych.

Podsumowując, klasy oferują większą elastyczność, rozszerzalność i możliwości niż struktury. Są bardziej odpowiednie w przypadkach, gdy potrzebujemy zarządzania stanem, dziedziczenia, polimorfizmu i bardziej zaawansowanych funkcji. Jednakże, ostateczny wybór między strukturami a klasami zależy od konkretnego przypadku i wymagań projektu.

Vector3

Oto lista funkcji i metod dostępnych w klasie Vector3 w Unity, wraz z przykładem użycia:

1. Magnitude:
   ```csharp
   float length = vector.magnitude;
   ```

2. Normalize:
   ```csharp
   Vector3 normalizedVector = vector.normalized;
   ```

3. Dot:
   ```csharp
   float dotProduct = Vector3.Dot(vector1, vector2);
   ```

4. Cross:
   ```csharp
   Vector3 crossProduct = Vector3.Cross(vector1, vector2);
   ```

5. Distance:
   ```csharp
   float distance = Vector3.Distance(point1, point2);
   ```

6. Lerp:
   ```csharp
   Vector3 interpolatedVector = Vector3.Lerp(startVector, endVector, t);
   ```

7. Project:
   ```csharp
   Vector3 projectedVector = Vector3.Project(vector, normal);
   ```

8. Reflect:
   ```csharp
   Vector3 reflectedVector = Vector3.Reflect(incomingVector, normal);
   ```

9. Angle:
   ```csharp
   float angle = Vector3.Angle(vector1, vector2);
   ```

10. Slerp:
    ```csharp
    Vector3 interpolatedVector = Vector3.Slerp(startVector, endVector, t);
    ```

11. Add:
    ```csharp
    Vector3 sum = vector1 + vector2;
    ```

12. Subtract:
    ```csharp
    Vector3 difference = vector1 - vector2;
    ```

13. Multiply:
    ```csharp
    Vector3 multipliedVector = vector * scalar;
    ```

14. Divide:
    ```csharp
    Vector3 dividedVector = vector / scalar;
    ```

15. Min:
    ```csharp
    Vector3 minVector = Vector3.Min(vector1, vector2);
    ```

16. Max:
    ```csharp
    Vector3 maxVector = Vector3.Max(vector1, vector2);
    ```

17. Clamp:
    ```csharp
    Vector3 clampedVector = Vector3.Clamp(vector, min, max);
    ```

18. MoveTowards:
```csharp
Vector3 newPosition = Vector3.MoveTowards(currentPosition, targetPosition, speed);
```

19. RotateTowards:
```csharp
Vector3 newDirection = Vector3.RotateTowards(currentDirection, targetDirection, rotationSpeed, 0);
```

20. Scale:
```csharp
Vector3 scaledVector = Vector3.Scale(vector, scale);
```

21. Round:
```csharp
Vector3 roundedVector = Vector3.Round(vector);
```

To tylko kilka przykładów funkcji i metod dostępnych w klasie Vector3 w Unity. Mogą być one używane do manipulacji i operacji na trójwymiarowych wektorach w celu tworzenia interakcji, ruchu, animacji i wielu innych funkcjonalności w grach i aplikacjach trójwymiarowych.

Quaternion

Quaternion jest strukturą w Unity używaną do reprezentowania rotacji w trójwymiarowej przestrzeni. Jest to bardziej zaawansowana metoda reprezentacji rotacji niż kąty Eulera czy macierze obracające. Quaternion składa się z czterech składowych: x, y, z i w, które są interpretowane jako osie i kąty obrotu.

Quaternions oferuje wiele przydatnych funkcji i operacji, które umożliwiają płynne interpolacje pomiędzy rotacjami, tworzenie rotacji na podstawie kierunków i punktów, wykonywanie obliczeń dotyczących rotacji oraz przekształcanie punktów w przestrzeni 3D przy zachowaniu poprawności rotacji.

Quaterniony są szczególnie przydatne w animacji, grach komputerowych i grafice 3D, gdzie płynne i poprawne obroty obiektów są istotne. Dzięki ich własnościom algebraicznym i możliwościom interpolacji, quaterniony umożliwiają płynne poruszanie się obiektów w przestrzeni, obracanie kamery, animację postaci, efekty specjalne i wiele innych.

Ważne jest zrozumienie, że Quaternion nie jest bezpośrednio czytelny dla człowieka, ponieważ reprezentuje kombinację osi i kątów obrotu. Jednak dzięki wbudowanym funkcjom i operacjom dostępnym w klasie Quaternion w Unity, można łatwo manipulować i operować na rotacjach w sposób intuicyjny i efektywny.

Oto lista funkcji i metod dostępnych w klasie Quaternion w Unity, wraz z przykładem użycia:

1. identity:
```csharp
Quaternion identityQuaternion = Quaternion.identity;
```

2. Euler:
```csharp
Quaternion eulerQuaternion = Quaternion.Euler(x, y, z);
```

3. LookRotation:
```csharp
Quaternion lookRotation = Quaternion.LookRotation(forward, upwards);
```

4. AngleAxis:
```csharp
Quaternion angleAxisQuaternion = Quaternion.AngleAxis(angle, axis);
```

5. Dot:
```csharp
float dotProduct = Quaternion.Dot(quaternion1, quaternion2);
```

6. Slerp:
```csharp
Quaternion interpolatedQuaternion = Quaternion.Slerp(startQuaternion, endQuaternion, t);
```

7. Lerp:
```csharp
Quaternion interpolatedQuaternion = Quaternion.Lerp(startQuaternion, endQuaternion, t);
```

8. Inverse:
```csharp
Quaternion inverseQuaternion = Quaternion.Inverse(quaternion);
```

9. Normalize:
```csharp
Quaternion normalizedQuaternion = Quaternion.Normalize(quaternion);
```

10. RotateTowards:
```csharp
Quaternion newRotation = Quaternion.RotateTowards(currentRotation, targetRotation, maxDegreesDelta);
```

11. FromToRotation:
```csharp
Quaternion fromToRotation = Quaternion.FromToRotation(fromDirection, toDirection);
```

12. EulerAngles:
```csharp
Vector3 eulerAngles = quaternion.eulerAngles;
```

13. Angle:
```csharp
float angle = Quaternion.Angle(quaternion1, quaternion2);
```

14. Equals:
```csharp
bool areEqual = quaternion1.Equals(quaternion2);
```

15. ToString:
```csharp
string quaternionString = quaternion.ToString();
```

16. operator *
```csharp
Quaternion multipliedQuaternion = quaternion1 * quaternion2;
```

17. operator ==
```csharp
bool areEqual = quaternion1 == quaternion2;
```

18. operator !=
```csharp
bool areNotEqual = quaternion1 != quaternion2;
```

To tylko kilka przykładów funkcji i metod dostępnych w klasie Quaternion w Unity. Quaternion jest używany do reprezentacji rotacji w trójwymiarowej przestrzeni i jest często stosowany w manipulacji obiektami w grach, animacjach, efektach specjalnych i innych interakcjach trójwymiarowych.

Get

Oto lista najpopularniejszych funkcji w Unity służących do pobierania komponentów i obiektów, wraz z opisami i przykładami użycia:

1. Find:
   Opis: Szuka obiektu w hierarchii sceny na podstawie nazwy.
   Przykład użycia:
   ```csharp
   GameObject foundObject = GameObject.Find("NazwaObiektu");
   ```

2. GetComponent:
   Opis: Pobiera komponent o określonym typie przypisanym do danego obiektu.
   Przykład użycia:
   ```csharp
   Rigidbody rb = GetComponent<Rigidbody>();
   ```

3. GetComponentInChildren:
   Opis: Pobiera komponent o określonym typie przypisanym do danego obiektu lub jego potomków.
   Przykład użycia:
   ```csharp
   Collider col = GetComponentInChildren<Collider>();
   ```

4. GetComponentInParent:
   Opis: Pobiera komponent o określonym typie przypisanym do danego obiektu lub jego rodzica.
   Przykład użycia:
   ```csharp
   AudioSource audioSource = GetComponentInParent<AudioSource>();
   ```

5. GetComponents:
   Opis: Pobiera wszystkie komponenty o określonym typie przypisane do danego obiektu.
   Przykład użycia:
   ```csharp
   Renderer[] renderers = GetComponents<Renderer>();
   ```

6. GetComponentsInChildren:
   Opis: Pobiera wszystkie komponenty o określonym typie przypisane do danego obiektu lub jego potomków.
   Przykład użycia:
   ```csharp
   Light[] lights = GetComponentsInChildren<Light>();
   ```

7. GetComponentsInParent:
   Opis: Pobiera wszystkie komponenty o określonym typie przypisane do danego obiektu lub jego rodzica.
   Przykład użycia:
   ```csharp
   Collider[] colliders = GetComponentsInParent<Collider>();
   ```

8. FindObjectOfType:
   Opis: Szuka obiektu w scenie o określonym typie komponentu.
   Przykład użycia:
   ```csharp
   GameManager gameManager = FindObjectOfType<GameManager>();
   ```

9. FindObjectsOfType:
   Opis: Szuka wszystkich obiektów w scenie o określonym typie komponentu.
   Przykład użycia:
   ```csharp
   Enemy[] enemies = FindObjectsOfType<Enemy>();
   ```

Te funkcje są powszechnie używane w Unity do pobierania i manipulowania komponentami oraz obiektami w scenie. Umożliwiają łatwe odnajdywanie, dostęp i interakcję z elementami gry, co jest niezwykle przydatne w programowaniu gier i tworzeniu interaktywnych doświadczeń.

REST

REST (Representational State Transfer) to architektura oprogramowania, która definiuje zestaw zasad i konwencji, które umożliwiają komunikację między systemami informatycznymi. W kontekście Unity, REST jest często wykorzystywany do komunikacji z serwerami i wymiany danych pomiędzy grą a zewnętrznymi usługami sieciowymi.

Unity zapewnia kilka narzędzi i bibliotek, które ułatwiają implementację komunikacji REST w grach. Jednym z najpopularniejszych narzędzi jest UnityWebRequest, który umożliwia wysyłanie żądań HTTP (GET, POST, PUT, DELETE itp.) do serwerów RESTful i odbieranie odpowiedzi.

Korzystając z UnityWebRequest, można łatwo komunikować się z serwerem, przesyłać dane w formacie JSON lub innych popularnych formatach, obsługiwać autoryzację, zarządzać nagłówkami HTTP i obsługiwać różne rodzaje odpowiedzi, takie jak tekst, obrazy, dźwięk itp.

Przykład użycia UnityWebRequest w celu pobrania danych z serwera RESTful:
```csharp
IEnumerator GetPlayerData()
{
    using (UnityWebRequest www = UnityWebRequest.Get("https://api.example.com/players/1"))
    {
        yield return www.SendWebRequest();

        if (www.result == UnityWebRequest.Result.Success)
        {
            string data = www.downloadHandler.text;
            // Przetwarzanie pobranych danych
        }
        else
        {
            Debug.Log("Błąd pobierania danych: " + www.error);
        }
    }
}
```

W powyższym przykładzie używamy UnityWebRequest do wysłania żądania GET na adres URL "https://api.example.com/players/1" w celu pobrania danych gracza. Odpowiedź serwera jest obsługiwana w zależności od wyniku żądania.

REST w Unity umożliwia integrację gier z różnymi usługami sieciowymi, takimi jak bazy danych, serwisy autoryzacyjne, serwisy zarządzania użytkownikami, pobieranie zasobów multimedialnych i wiele innych. Daje to twórcom gier możliwość tworzenia interaktywnych i dynamicznych doświadczeń, które mogą korzystać z danych i funkcjonalności dostępnych na zewnętrznych serwerach.

Oto kilka ważnych funkcji i metod w Unity związanych z REST:

1. UnityWebRequest.Get:
   Metoda służąca do wysyłania żądania HTTP GET na określony URL.
   Przykład użycia:
   ```csharp
   UnityWebRequest www = UnityWebRequest.Get("https://api.example.com/data");
   ```

2. UnityWebRequest.Post:
   Metoda służąca do wysyłania żądania HTTP POST na określony URL.
   Przykład użycia:
   ```csharp
   UnityWebRequest www = UnityWebRequest.Post("https://api.example.com/data", form);
   ```

3. UnityWebRequest.Put:
   Metoda służąca do wysyłania żądania HTTP PUT na określony URL.
   Przykład użycia:
   ```csharp
   UnityWebRequest www = UnityWebRequest.Put("https://api.example.com/data", jsonData);
   ```

4. UnityWebRequest.Delete:
   Metoda służąca do wysyłania żądania HTTP DELETE na określony URL.
   Przykład użycia:
   ```csharp
   UnityWebRequest www = UnityWebRequest.Delete("https://api.example.com/data/1");
   ```

5. SetRequestHeader:
   Metoda służąca do ustawiania nagłówków HTTP w żądaniu.
   Przykład użycia:
   ```csharp
   www.SetRequestHeader("Authorization", "Bearer <access_token>");
   ```

6. SendWebRequest:
   Metoda służąca do wysłania żądania do serwera i oczekiwania na odpowiedź.
   Przykład użycia:
   ```csharp
   yield return www.SendWebRequest();
   ```

7. responseCode:
   Właściwość zawierająca kod odpowiedzi HTTP z serwera.
   Przykład użycia:
   ```csharp
   if (www.responseCode == 200)
   {
      // Odpowiedź sukcesu
   }
   ```

8. downloadHandler:
   Właściwość zawierająca obiekt odpowiedzialny za obsługę pobierania danych z odpowiedzi.
   Przykład użycia:
   ```csharp
   byte[] responseData = www.downloadHandler.data;
   ```

9. uploadHandler:
   Właściwość zawierająca obiekt odpowiedzialny za przesyłanie danych w żądaniu.
   Przykład użycia:
   ```csharp
   www.uploadHandler = new UploadHandlerRaw(requestData);
   ```

10. isNetworkError:
    Właściwość wskazująca, czy wystąpił błąd sieciowy podczas wysyłania żądania.
    Przykład użycia:
    ```csharp
    if (www.isNetworkError)
    {
       Debug.Log("Błąd sieciowy: " + www.error);
    }
    ```

Te funkcje i metody dostarczają podstawowe narzędzia do komunikacji z serwerami RESTful w Unity. Pozwalają na wysyłanie różnych rodzajów żądań HTTP, zarządzanie nagłówkami, obsługę danych odpowiedzi oraz obsługę błędów sieciowych. Możesz je wykorzyst

ać do integracji z różnymi usługami sieciowymi i wymiany danych między grą Unity a zewnętrznymi serwerami.

Try-catch-finally

Try-catch-finally to konstrukcja językowa używana w programowaniu do obsługi wyjątków i zapewnienia odpowiednich działań w przypadku wystąpienia błędu. Składa się z trzech bloków: try, catch i finally.

1. Blok try:
   Blok try zawiera kod, który może potencjalnie generować wyjątki. Jest to miejsce, w którym umieszczamy kod, który chcemy monitorować pod kątem wyjątków. Jeśli w trakcie wykonywania kodu w bloku try wystąpi wyjątek, program przejdzie do odpowiedniego bloku catch.

Przykład:
```csharp
try
{
    // Kod, który może generować wyjątki
}
```

2. Blok catch:
Blok catch służy do przechwytywania i obsługi wyjątków. W tym bloku określamy, jaki rodzaj wyjątku chcemy przechwycić, a następnie definiujemy działania, które mają zostać wykonane w przypadku wystąpienia tego wyjątku.

Przykład:
```csharp
catch (Exception ex)
{
    // Obsługa wyjątku
}
```

W powyższym przykładzie `Exception` jest ogólnym typem wyjątku, który przechwytuje wszystkie rodzaje wyjątków. Możemy także użyć bardziej konkretnej klasy wyjątku, np. `ArgumentException`, aby przechwycić tylko wyjątki tego typu.

3. Blok finally:
Blok finally zawiera kod, który zostanie wykonany niezależnie od tego, czy wystąpił wyjątek czy nie. Ten blok jest opcjonalny. Często używany jest do wykonania czynności czyszczących lub zwolnienia zasobów.

Przykład:
```csharp
finally
{
    // Kod, który zawsze zostanie wykonany
}
```

Blok finally jest przydatny, ponieważ gwarantuje, że pewne czynności zostaną wykonane niezależnie od tego, czy wystąpił wyjątek, czy nie.

Dodatkowo, w C# dostępne są również inne bloki obsługi wyjątków, takie jak:

- Blok catch z wieloma klauzulami: Pozwala na przechwycenie różnych rodzajów wyjątków w jednym bloku catch.

Przykład:
```csharp
catch (ExceptionType1 ex1)
{
    // Obsługa wyjątku typu ExceptionType1
}
```

```
catch (ExceptionType2 ex2)
{
    // Obsługa wyjątku typu ExceptionType2
}
```

- Blok finally zabezpieczony: Pozwala na zdefiniowanie kodu, który zostanie wykonany po zakończeniu bloku try, niezależnie od tego, czy wystąpił wyjątek, czy nie.

Przykład:
```csharp
try
{
    // Kod
}
finally
{


// Kod, który zawsze zostanie wykonany po zakończeniu bloku try
}
```

Wszystkie te bloki są używane w celu zapewnienia prawidłowej obsługi wyjątków i kontrolowanego działania programu nawet w przypadku nieprzewidzianych sytuacji.

Rzutowanie typów

Rzutowanie typów, znane również jako konwersja typów, odnosi się do procesu konwertowania jednego typu danych na inny typ danych. W języku C# istnieją dwa rodzaje rzutowania: rzutowanie jawne (explicit casting) i rzutowanie niejawne (implicit casting).

1. Rzutowanie jawne (explicit casting):
Rzutowanie jawne jest wymuszone przez programistę i jest stosowane, gdy konwersja między typami może spowodować utratę danych lub potencjalne błędy. W tym przypadku programista musi jawnie określić typ docelowy.

Przykład:
```csharp
double x = 3.14;
int y = (int)x; // Rzutowanie jawne
```

W powyższym przykładzie zmienna `x` jest rzutowana na typ `int`, co oznacza, że wartość po przecinku zostanie utracona.

2. Rzutowanie niejawne (implicit casting):
Rzutowanie niejawne jest automatyczne i nie wymaga jawnego określenia typu docelowego. Jest stosowane, gdy konwersja między typami nie powoduje utraty danych lub potencjalnych błędów.

Przykład:
```csharp
int a = 10;
double b = a; // Rzutowanie niejawne
```

W powyższym przykładzie zmienna `a` o typie `int` jest automatycznie rzutowana na typ `double`.

Warto pamiętać, że nie każde rzutowanie jest możliwe. Rzutowanie może być wykonywane tylko między typami, które są ze sobą zgodne. Na przykład, nie można rzutować typu `string` na `int`, chyba że wartość w ciągu znaków można przekonwertować na liczbę całkowitą.

Rzutowanie typów jest przydatne, gdy chcemy przekonwertować wartości między różnymi typami danych w celu wykonania operacji matematycznych, porównań lub przypisań. Jednak należy zachować ostrożność podczas używania rzutowania, aby uniknąć utraty danych lub nieprawidłowych wyników.

W przypadku pracy z komponentami w Unity, rzutowanie jest często używane do uzyskania dostępu do konkretnych komponentów i operowania nimi. Oto przykłady rzutowania typów komponentów w Unity:

1. Rzutowanie na komponenty MonoBehaviour:
Gdy chcemy uzyskać dostęp do komponentów dziedziczących z klasy `MonoBehaviour`, możemy użyć metody `GetComponent<T>()`, która zwraca komponent o określonym typie. Następnie możemy dokonać rzutowania na konkretny typ komponentu, aby móc używać jego metod i właściwości.

Przykład:
```csharp
Rigidbody2D rb2d = GetComponent<Rigidbody2D>(); // Pobranie komponentu Rigidbody2D
rb2d.AddForce(Vector2.up * 10f); // Wywołanie metody na skomponowanym Rigidbody2D
```

2. Rzutowanie na komponenty specjalne:
Unity dostarcza specjalne komponenty, takie jak `Transform`, `Collider`, `AudioSource` itp., które są szeroko używane w tworzeniu gier. Rzutowanie na te komponenty jest często wykorzystywane do manipulacji ich właściwościami lub wywoływania odpowiednich metod.

Przykład:
```csharp
Transform playerTransform = transform; // Rzutowanie na komponent Transform
playerTransform.position = new Vector3(0f, 0f, 0f); // Ustawienie pozycji obiektu
```

3. Rzutowanie na interfejsy:
Często używamy interfejsów w Unity do definiowania wspólnych zachowań dla różnych komponentów. Możemy dokonać rzutowania na interfejs, aby uzyskać dostęp do metod i właściwości zdefiniowanych w interfejsie.

Przykład:
```csharp
IInteractable interactable = GetComponent<IInteractable>(); // Rzutowanie na interfejs IInteractable
interactable.Interact(); // Wywołanie metody z interfejsu
```

Rzutowanie typów komponentów jest powszechnie stosowane w Unity, aby uzyskać dostęp do funkcjonalności komponentów i manipulować nimi w trakcie działania gry. Ważne jest, aby upewnić się, że rzutowanie jest prawidłowe i nie spowoduje błędów wykonania.

Deserializacja

Deserializacja odnosi się do procesu konwertowania danych w postaci zserializowanej, takiej jak JSON lub XML, z powrotem na oryginalne obiekty lub struktury danych. Jest to często stosowane w programowaniu do wczytywania danych zapisanych w plikach lub przesyłanych przez sieć.

W języku C# i w środowisku Unity istnieje wiele sposobów deserializacji danych. Oto kilka popularnych narzędzi i technik deserializacji:

1. JsonUtility:
W Unity można użyć klasy `JsonUtility` do deserializacji danych z formatu JSON. Klasa ta udostępnia metody, takie jak `JsonUtility.FromJson()`, które konwertują dane JSON na obiekty C#.

Przykład:
```csharp
string json = "{\"name\":\"John\",\"age\":30}";
PlayerData playerData = JsonUtility.FromJson<PlayerData>(json);
```

2. XML Serialization:
W przypadku danych w formacie XML można użyć mechanizmu serializacji i deserializacji wbudowanego w język C#. Można to osiągnąć za pomocą atrybutów takich jak `[Serializable]` i `[XmlElement]`, oraz przy użyciu klas takich jak `XmlSerializer` i `XmlDocument`.

Przykład:
```csharp
XmlSerializer serializer = new XmlSerializer(typeof(PlayerData));
using (StreamReader reader = new StreamReader("data.xml"))
{
    PlayerData playerData = (PlayerData)serializer.Deserialize(reader);
}
```

3. Binary Serialization:
Jeśli dane są w formacie binarnym, można użyć mechanizmu serializacji binarnej w języku C#. Można to osiągnąć za pomocą klas takich jak `BinaryFormatter` i `MemoryStream`.

Przykład:
```csharp
BinaryFormatter formatter = new BinaryFormatter();
using (FileStream stream = new FileStream("data.bin", FileMode.Open))
{
    PlayerData playerData = (PlayerData)formatter.Deserialize(stream);
}
```

Deserializacja jest przydatna, gdy chcemy przywrócić dane w formacie zserializowanym do oryginalnej postaci obiektów lub struktur danych. Umożliwia to wczytywanie danych z plików, sieci lub innych źródeł i używanie ich w aplikacji lub grze. Ważne jest, aby dbać o poprawność deserializacji i sprawdzić, czy dane wejściowe są zgodne z oczekiwanym formatem, aby uniknąć błędów i niezgodności.

Serializacja

Serializacja odnosi się do procesu konwersji obiektów lub struktur danych na postać, która może być zapisana lub przesłana, na przykład do pliku lub przez sieć. Jest to przydatne w programowaniu, gdy chcemy zachować dane w trwałej formie lub przesłać je do innej aplikacji lub systemu.

W języku C# i w środowisku Unity istnieje wiele sposobów serializacji danych. Oto kilka popularnych narzędzi i technik serializacji:

1. JsonUtility:
   W Unity można użyć klasy `JsonUtility` do serializacji danych do formatu JSON. Klasa ta udostępnia metody, takie jak `JsonUtility.ToJson()`, które konwertują obiekty C# na dane JSON.

   Przykład:
   ```csharp
   PlayerData playerData = new PlayerData("John", 30);
   string json = JsonUtility.ToJson(playerData);
   ```

2. XML Serialization:
   W przypadku danych w formacie XML można użyć mechanizmu serializacji i deserializacji wbudowanego w język C#. Można to osiągnąć za pomocą atrybutów takich jak `[Serializable]` i `[XmlElement]`, oraz przy użyciu klas takich jak `XmlSerializer` i `XmlDocument`.

   Przykład:
   ```csharp
   XmlSerializer serializer = new XmlSerializer(typeof(PlayerData));
   using (StreamWriter writer = new StreamWriter("data.xml"))
   {
       serializer.Serialize(writer, playerData);
   }
   ```

3. Binary Serialization:
   Można również użyć mechanizmu serializacji binarnej w języku C#, aby przekształcić obiekty w formę binarną. Klasa `BinaryFormatter` jest często wykorzystywana do tego celu.

   Przykład:
   ```csharp
   BinaryFormatter formatter = new BinaryFormatter();
   using (FileStream stream = new FileStream("data.bin", FileMode.Create))
   {
      formatter.Serialize(stream, playerData);
   }
   ```

Serializacja jest używana do przechowywania danych w formacie zserializowanym, który można łatwo zapisać, przesłać lub odczytać w innych miejscach. Pozwala to na zachowanie stanu obiektów, przenoszenie danych między aplikacjami lub komunikację z serwerami. Ważne jest, aby pamiętać o zabezpieczaniu danych przed serializacją i deserializacją oraz o uwzględnieniu wersji danych, aby uniknąć problemów niezgodności w przyszłości.

Oto przykład serializacji klasy niestandardowej w Unity, wykorzystujący mechanizm serializacji JSON za pomocą klasy `JsonUtility`:

```csharp
using UnityEngine;

[System.Serializable]
public class PlayerData
{
   public string playerName;
   public int playerLevel;

   public PlayerData(string name, int level)
   {
      playerName = name;
      playerLevel = level;
   }
}

public class SerializationExample : MonoBehaviour
{
   private void Start()
   {
      // Tworzenie obiektu PlayerData
      PlayerData playerData = new PlayerData("John", 30);

      // Serializacja do formatu JSON
      string json = JsonUtility.ToJson(playerData);

      // Zapisanie do pliku
      System.IO.File.WriteAllText("playerData.json", json);

      Debug.Log("Serializacja zakończona. Zapisano dane do pliku.");
```

```
        // Odczytanie z pliku

        // Odczytanie danych z pliku
        string jsonLoaded = System.IO.File.ReadAllText("playerData.json");

        // Deserializacja z formatu JSON
        PlayerData loadedPlayerData = JsonUtility.FromJson<PlayerData>(jsonLoaded);

        Debug.Log("Deserializacja zakończona. Wczytano dane z pliku.");

        // Wyświetlenie wczytanych danych
        Debug.Log("Player Name: " + loadedPlayerData.playerName);
        Debug.Log("Player Level: " + loadedPlayerData.playerLevel);
    }
}
```

W powyższym przykładzie tworzymy klasę `PlayerData`, która przechowuje informacje o graczu (nazwa i poziom). Klasa ta jest oznaczona atrybutem `[System.Serializable]`, co umożliwia jej serializację.

W metodzie `Start()` tworzymy obiekt `PlayerData` o wartościach "John" i 30, a następnie serializujemy go do formatu JSON za pomocą `JsonUtility.ToJson()`. Następnie zapisujemy dane do pliku "playerData.json" przy użyciu `System.IO.File.WriteAllText()`.

Następnie odczytujemy dane z pliku za pomocą `System.IO.File.ReadAllText()` i deserializujemy je z formatu JSON przy użyciu `JsonUtility.FromJson()`. Wczytane dane są przechowywane w nowym obiekcie `loadedPlayerData`, który możemy wykorzystać w dalszej części aplikacji.

Na koniec wyświetlamy wczytane dane za pomocą `Debug.Log()` w celu weryfikacji, czy deserializacja została wykonana poprawnie.

Ten przykład ilustruje proces serializacji i deserializacji niestandardowej klasy w Unity przy użyciu `JsonUtility`.

Atrybuty

Atrybuty (Attributes) są specjalnymi metadanymi, które można przypisywać do elementów kodu w języku C#. Atrybuty dostarczają dodatkowych informacji o danym elemencie i mogą wpływać na sposób jego kompilacji, działania lub zachowania w czasie wykonania. Są one wykorzystywane do wzbogacania kodu o metadane, które mogą być odczytywane i wykorzystywane przez inne narzędzia, frameworki lub środowiska.

Oto kilka przykładów popularnych atrybutów w języku C#:

1. `[Serializable]` - Ten atrybut jest używany do oznaczenia klasy, która ma być serializowana. Umożliwia konwersję obiektu klasy na postać, która może być zapisana lub przesłana, na przykład do formatu JSON lub XML.

2. `[Obsolete]` - Atrybut `[Obsolete]` służy do oznaczania elementów kodu, które są przestarzałe i powinny być zastąpione nowymi. Pomaga w świadomej migracji do nowych interfejsów lub funkcji.

3. `[Conditional]` - Atrybut `[Conditional]` jest stosowany do metod, które mają być wywoływane warunkowo w zależności od obecności określonego symbolu kompilacji. Pozwala na kompilację i uruchomienie kodu w różnych konfiguracjach.

4. `[DllImport]` - Ten atrybut jest używany w kontekście interop z kodem natywnym. Wskazuje, że określona metoda jest importowana z biblioteki dynamicznej.

5. `[SerializeField]` - Atrybut `[SerializeField]` jest używany w Unity w kontekście serializacji pól prywatnych. Pozwala na serializację i wyświetlanie prywatnych pól w inspektorze Unity.

Przykład użycia atrybutu `[SerializeField]` w Unity:

```csharp
public class Player : MonoBehaviour
{
    [SerializeField]
    private int health;

    [SerializeField]
    private string playerName;

    // Reszta kodu...
}
```

W powyższym przykładzie atrybut `[SerializeField]` jest używany do oznaczenia prywatnych pól `health` i `playerName` w klasie `Player`. Dzięki temu pola te będą widoczne i edytowalne w inspektorze Unity, co umożliwi łatwe ustawianie wartości w edytorze.

Atrybuty są potężnym narzędziem w języku C#, które pozwalają na dostarczanie dodatkowych informacji o kodzie i wpływanie na jego zachowanie. Dzięki nim można osiągnąć większą elastyczność i dostosowanie kodu do różnych wymagań i narzędzi.

Properties

W języku C#, właściwości (properties) są specjalnymi elementami składni, które pozwalają na definiowanie getterów i/lub setterów dla dostępu do pól klasy. Właściwości umożliwiają kontrolę nad tym, jak wartości są odczytywane i zapisywane, oraz mogą dodawać dodatkową logikę przy pobieraniu lub ustawianiu wartości.

Właściwości są często używane jako publiczne interfejsy do dostępu do pól prywatnych klasy, co umożliwia kontrolę nad tym, jak te pola są odczytywane i zmieniane. Oto przykład deklaracji właściwości w języku C#:

```csharp
public class Person
{
    private string name;
    private int age;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public int Age
    {
        get { return age; }
        set { age = value; }
    }
}
```

W powyższym przykładzie klasa `Person` posiada dwie właściwości: `Name` i `Age`. Każda z tych właściwości ma zdefiniowane bloki gettera i settera. Getter zwraca wartość pola, a setter ustawia wartość pola na podstawie przekazanego argumentu.

Przykład użycia:

```csharp
Person person = new Person();
person.Name = "John";
person.Age = 30;

Console.WriteLine("Name: " + person.Name);
Console.WriteLine("Age: " + person.Age);
```

Wynik:

```
Name: John
Age: 30
```

Dzięki użyciu właściwości, dostęp do pól klasy `Person` odbywa się poprzez interfejs gettera i/lub settera, co umożliwia kontrolę nad operacjami odczytu i zapisu. Właściwości są bardziej elastyczną i bezpieczną alternatywą dla bezpośredniego dostępu do pól, ponieważ umożliwiają dodanie dodatkowej logiki, walidacji i kontroli przy pobieraniu i ustawianiu wartości.

Propery w Unity

W Unity właściwości (properties) są szeroko wykorzystywane do zarządzania danymi i dostępu do nich w komponentach i obiektach gry. Właściwości w Unity mają wiele zastosowań, takich jak dostęp do wartości parametrów, kontroli widoczności w inspektorze, synchronizacji z innymi komponentami, obsługi zdarzeń itp.

Oto kilka przykładów zastosowania właściwości w Unity:

1. Dostęp do danych komponentu:
```csharp
public class Player : MonoBehaviour
{
    private int health;

    public int Health
    {
        get { return health; }
        set { health = value; }
    }
}
```
W powyższym przykładzie, właściwość `Health` pozwala na dostęp do prywatnego pola `health` klasy `Player`. Dzięki temu można kontrolować odczyt i zapis wartości `health`, a także wykonywać dodatkową logikę w blokach gettera i settera.

2. Kontrola widoczności w inspektorze Unity:
```csharp
public class Player : MonoBehaviour
{
    [SerializeField]
    private int score;

    public int Score
    {
        get { return score; }
        private set { score = value; }
    }
}
```
W tym przypadku, używając atrybutu `[SerializeField]`, pola `score` będą widoczne w inspektorze Unity, ale dzięki użyciu właściwości `Score` można zapewnić, że wartość pola będzie tylko do odczytu z zewnątrz, a nie można jej zmieniać z poziomu innych skryptów.

3. Synchronizacja danych z innymi komponentami:
```csharp
public class HealthDisplay : MonoBehaviour
{
    [SerializeField]
    private Player player;

    private Text healthText;

    private void Start()
    {
        healthText = GetComponent<Text>();
    }

    private void Update()
    {
        healthText.text = "Health: " + player.Health.ToString();
    }
}
```
W tym przykładzie, skrypt `HealthDisplay` posiada referencję do komponentu `Player`. Za pomocą właściwości `Health` w `Player`, można odczytywać wartość `Health` i synchronizować ją z innymi elementami interfejsu, takimi jak tekstowy komponent `Text`, aby wyświetlać aktualny stan zdrowia gracza.

Właściwości w Unity dają kontrolę nad dostępem do danych, pozwalając na implementację logiki i zabezpieczeń, a także ułatwiają interakcję między komponentami i obiektami gry.

Static

W języku C# słowo kluczowe "static" jest używane do deklaracji składowych (metod, pól, właściwości, zagnieżdżonych klas) jako elementów statycznych, czyli przypisanych do samej klasy, a nie do instancji klasy. Oznacza to, że składowa statyczna istnieje tylko w jednym egzemplarzu niezależnie od liczby utworzonych instancji danej klasy.

Oto kilka cech i zastosowań składowych statycznych:

1. Brak konieczności tworzenia instancji: Składowe statyczne mogą być wywoływane bez konieczności tworzenia obiektu klasy. Można uzyskać do nich dostęp bezpośrednio poprzez nazwę klasy.

2. Dostęp do składowych statycznych: Można uzyskać dostęp do składowych statycznych za pomocą operatora kropki (.), korzystając z nazwy klasy, a nie konkretnej instancji klasy.

3. Współdzielenie wartości: Składowe statyczne są współdzielone przez wszystkie instancje klasy. Oznacza to, że zmiana wartości składowej statycznej będzie miała wpływ na wszystkie obiekty klasy.

4. Przechowywanie danych globalnych: Składowe statyczne często są używane do przechowywania danych globalnych, które mają być dostępne i modyfikowalne w różnych miejscach w ramach aplikacji.

5. Metody pomocnicze: Metody statyczne są często używane do implementacji funkcji pomocniczych, które nie wymagają dostępu do stanu obiektu i nie operują na danych instancji.

Przykład zastosowania składowych statycznych:

```csharp
public class MathUtils
{
   public static int Multiply(int a, int b)
   {
      return a * b;
   }
}

// Wywołanie metody statycznej bez tworzenia instancji klasy MathUtils
int result = MathUtils.Multiply(2, 3);
```

W powyższym przykładzie, metoda statyczna `Multiply` w klasie `MathUtils` może być wywoływana bez potrzeby tworzenia instancji klasy. Używając składnika statycznego, można bezpośrednio uzyskać dostęp do tej metody i obliczyć wynik mnożenia dwóch liczb.

Składowe statyczne są przydatne w przypadkach, gdy nie jest wymagane tworzenie i zarządzanie instancjami klasy, a potrzebne jest wykonanie pewnych operacji globalnych lub udostępnienie funkcji pomocniczych bezpośrednio na poziomie klasy.

W Unity, słowo kluczowe "static" może być używane do deklaracji zmiennych i klas statycznych. Oto krótkie wyjaśnienie obu przypadków:

1. Statyczne zmienne w Unity:
   - Statyczne zmienne są zmiennymi, które są przypisane do klasy, a nie do instancji klasy. Istnieje tylko jedna kopia statycznej zmiennej, niezależnie od liczby utworzonych instancji klasy.
   - Statyczne zmienne są używane do przechowywania danych, które mają być współdzielone przez wszystkie instancje klasy lub dostępne globalnie w obrębie projektu.
   - Przykład:
     ```csharp
     public class GameManager : MonoBehaviour
     {
        public static int score;

        void Start()
        {
           score = 0;
        }
     }
     ```
     W tym przykładzie, statyczna zmienna `score` w klasie `GameManager` przechowuje wynik gry i jest dostępna do odczytu i zapisu z dowolnego miejsca w projekcie.

2. Statyczne klasy w Unity:
   - Statyczna klasa w Unity to klasa, której metody i pola są statyczne i można do nich uzyskać dostęp bez konieczności tworzenia instancji klasy.
   - Statyczne klasy są często używane do implementacji funkcji pomocniczych, narzędzi, narzędzi do zarządzania zasobami itp.
   - Przykład:
     ```csharp
     public static class MathUtils
     {
        public static int Add(int a, int b)
        {
           return a + b;
        }
     }
     ```
     W tym przykładzie, statyczna klasa `MathUtils` zawiera statyczną metodę `Add`, która wykonuje operację dodawania dwóch liczb. Metoda ta może być wywoływana bez tworzenia instancji klasy `MathUtils`.

Statyczne zmienne i klasy są przydatne w Unity do przechowywania danych globalnych, implementacji funkcji pomocniczych i udostępniania operacji, które nie wymagają dostępu do konkretnej instancji klasy. Jednak warto pamiętać, że nadmierne użycie składników statycznych może prowadzić do utraty kontroli nad stanem i utrudnienia testowania i zarządzania kodem. Dlatego ważne jest umiejętne stosowanie składników statycznych w zależności od potrzeb i zasad projektowych.

PlayerPrefs

PlayerPrefs w Unity to prosta klasa, która umożliwia przechowywanie danych w pamięci urządzenia, takich jak preferencje gracza, ustawienia, postępy w grze itp. Playerprefs jest często używany do zapisywania i odczytywania danych trwałych między sesjami gry. Oto podstawowe informacje na temat korzystania z PlayerPrefs:

- Zapisywanie danych:
  Aby zapisać dane w PlayerPrefs, użyj metody `SetInt`, `SetFloat`, `SetString` lub innych odpowiednich metod, w zależności od typu danych. Przykład:
  ```csharp
  PlayerPrefs.SetInt("HighScore", 1000);
  PlayerPrefs.SetFloat("Volume", 0.5f);
  PlayerPrefs.SetString("PlayerName", "John");
  PlayerPrefs.Save();
  ```
- Odczytywanie danych:
  Aby odczytać dane z PlayerPrefs, użyj metod `GetInt`, `GetFloat`, `GetString` lub innych odpowiednich metod, w zależności od typu danych. Przykład:
  ```csharp
  int highScore = PlayerPrefs.GetInt("HighScore");
  float volume = PlayerPrefs.GetFloat("Volume");
  string playerName = PlayerPrefs.GetString("PlayerName");
  ```

- Usuwanie danych:
  Aby usunąć dane z PlayerPrefs, użyj metody `DeleteKey` lub `DeleteAll`. Przykład:
  ```csharp
  PlayerPrefs.DeleteKey("HighScore"); // Usuwa pojedynczy klucz
  PlayerPrefs.DeleteAll(); // Usuwa wszystkie dane PlayerPrefs
  ```

Ważne jest, aby pamiętać, że PlayerPrefs przechowuje dane w prostym formacie klucz-wartość, więc nie jest odpowiednie do przechowywania dużych ilości danych lub złożonych struktur danych. Playerprefs jest również dostępne tylko na poziomie jednego urządzenia, co oznacza, że dane nie będą dostępne na innych urządzeniach. W przypadku przechowywania danych, które wymagają synchronizacji między różnymi urządzeniami, należy rozważyć inne rozwiązania, takie jak serwery lub usługi w chmurze.

PlayerPrefs jest przydatnym narzędziem do przechowywania prostych danych trwałych w Unity, takich jak preferencje gracza, ustawienia lub proste postępy w grze. Jednak dla bardziej zaawansowanych przypadków przechowywania danych lub wymagających synchronizacji między urządzeniami, należy rozważyć inne rozwiązania.

Wzorzec projektowy

Wzorzec projektowy (ang. design pattern) to sprawdzony, powszechnie akceptowany i powtarzalny sposób rozwiązania określonego problemu projektowego w programowaniu. Jest to ogólna, abstrakcyjna koncepcja, która może być zastosowana w różnych kontekstach i językach programowania.

Wzorce projektowe służą do udostępnienia wypróbowanych rozwiązań dla często spotykanych problemów projektowych. Pomagają programistom unikać powtarzania kodu, zwiększają czytelność i zrozumiałość kodu oraz ułatwiają utrzymanie i modyfikację aplikacji w przyszłości. Wzorce projektowe są oparte na doświadczeniach i najlepszych praktykach programistycznych, co przekłada się na bardziej efektywne i efektywne tworzenie oprogramowania.

Wzorce projektowe można podzielić na różne kategorie, takie jak wzorce kreacyjne, strukturalne i behawioralne. Przykłady popularnych wzorców projektowych to Singleton, Fabryka, Obserwator, Dekorator, Strategia, Iterator i wiele innych. Każdy wzorzec ma swoje unikalne zastosowanie i sposób działania, dostosowany do konkretnych problemów projektowych.

Wykorzystywanie wzorców projektowych pozwala programistom na korzystanie z gotowych rozwiązań, które zostały przetestowane i udowodnione w praktyce. Wzorce projektowe są narzędziem, które pomaga w tworzeniu elastycznego, modułowego i skalowalnego oprogramowania, które jest łatwe do zrozumienia i utrzymania.

Singleton

Singleton jest wzorcem projektowym, który służy do zapewnienia, że dana klasa ma tylko jedną instancję i zapewnia globalny punkt dostępu do tej instancji. Gwarantuje to, że niezależnie od ilości żądań, zawsze istnieje tylko jedna instancja danej klasy.

Główne cechy wzorca Singleton to:

1. Jedna instancja: Wzorzec Singleton zapewnia, że klasa ma tylko jedną instancję, która jest globalnie dostępna.

2. Globalny dostęp: Instancja Singletona jest dostępna globalnie i może być wywoływana z dowolnego miejsca w kodzie.

3. Leniwe inicjalizowanie: Instancja Singletona jest tworzona tylko wtedy, gdy jest potrzebna, a nie na etapie inicjalizacji programu.

4. Unikalność: Wzorzec Singleton zapewnia, że nie może istnieć więcej niż jedna instancja danej klasy.

Singleton jest często używany w sytuacjach, gdy istnienie tylko jednej instancji danej klasy jest kluczowe dla poprawnego działania systemu. Przykłady zastosowania Singletona to zarządzanie połączeniami do bazy danych, dostęp do plików konfiguracyjnych, rejestry aplikacji lub inne zasoby, które powinny być globalnie dostępne, ale nie powinny mieć wielu instancji. Implementacja wzorca Singleton polega na ukryciu konstruktora klasy i udostępnieniu statycznej metody, która zwraca jedyną instancję klasy. Przykład implementacji wzorca Singleton w języku C# może wyglądać tak:

```csharp
public class Singleton
{
    private static Singleton instance;

    // Prywatny konstruktor, aby uniemożliwić bezpośrednie tworzenie instancji
    private Singleton()
    {
        // Inicjalizacja Singletona
    }

    // Statyczna metoda, która zwraca instancję Singletona
    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }

    // Metody i właściwości klasy Singleton
}
```

Dzięki wzorcowi Singleton można mieć pewność, że dostęp do instancji klasy będzie zawsze kontrolowany i zapewniony tylko jedna instancja w całym systemie.

SOLID to zbiór zasad projektowania oprogramowania, które promują dobre praktyki tworzenia czytelnego, elastycznego i łatwego do utrzymania kodu. SOLID składa się z pięciu zasad, które są opisane poniżej:

1. Zasada pojedynczej odpowiedzialności (Single Responsibility Principle, SRP): Klasa powinna mieć tylko jedną odpowiedzialność i powinna być zmieniana tylko z powodu jednej przyczyny. Każda klasa powinna mieć jeden, klarowny cel i odpowiadać za jedną konkretną funkcjonalność.

2. Zasada otwarte-zamknięte (Open-Closed Principle, OCP): Klasa powinna być otwarta na rozszerzenia, ale zamknięta na modyfikacje. Oznacza to, że zmiany w zachowaniu klasy powinny być wprowadzane poprzez dodawanie nowego kodu, a nie przez modyfikację istniejącego kodu.

3. Zasada podstawienia Liskov (Liskov Substitution Principle, LSP): Obiekty danej klasy powinny być zastępowalne obiektami jej podklas, nie powodując nieoczekiwanych efektów ubocznych. Klasa pochodna powinna być w stanie zastąpić klasę bazową bez naruszania spodziewanego zachowania programu.

4. Zasada segregacji interfejsów (Interface Segregation Principle, ISP): Klienci nie powinni być zmuszani do zależności od interfejsów, których nie używają. Interfejsy powinny być konkretne, specjalizowane i dostosowane do potrzeb poszczególnych klientów.

5. Zasada odwrócenia zależności (Dependency Inversion Principle, DIP): Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Obiektom należy udostępniać abstrakcje, a nie konkretne implementacje. Zależności powinny być oparte na abstrakcjach, nie na szczegółach implementacyjnych.

Zasady SOLID pomagają tworzyć modularny, skalowalny i łatwo testowalny kod. Przestrzeganie tych zasad sprzyja elastyczności, rozszerzalności i utrzymaniu aplikacji w długim okresie czasu. Wprowadzając SOLID do swojej praktyki programistycznej, można zwiększyć jakość i trwałość swojego oprogramowania.

Lambda

Lambda to wyrażenie anonimowe w języku C#, które umożliwia definiowanie i używanie funkcji w miejscach, gdzie oczekiwane są delegaty lub wyrażenia funkcyjne. Wyrażenia lambda są skróconą i bardziej zwięzłą alternatywą dla tradycyjnych metod.

Składnia wyrażeń lambda składa się z parametrów wejściowych, strzałki "->" i ciała funkcji. Parametry wejściowe są deklarowane w nawiasach, a ciało funkcji jest umieszczone po strzałce.

Przykład składni wyrażenia lambda:
```csharp
(parametry) => ciało_funkcji
```

Wyrażenia lambda mogą być używane w różnych kontekstach, takich jak sortowanie listy, filtrowanie danych, wykonanie operacji na kolekcjach, obsługa zdarzeń itp.

Przykład użycia wyrażenia lambda w sortowaniu listy:
```csharp
List<int> numbers = new List<int> { 3, 1, 4, 2, 5 };
numbers.Sort((a, b) => a.CompareTo(b));
```

W powyższym przykładzie wyrażenie lambda jest używane jako argument do metody `Sort()` listy. Wyrażenie `(a, b) => a.CompareTo(b)` definiuje porównanie między dwoma elementami listy.

Biblioteka LINQ

Biblioteka LINQ (Language Integrated Query) jest częścią języka C# i .NET Framework, która zapewnia potężne narzędzia do zapytań i manipulacji różnymi źródłami danych. LINQ umożliwia programistom wykonywanie zaawansowanych operacji na kolekcjach, bazach danych, plikach XML i innych źródłach danych, korzystając z wyrażeń zapytań podobnych do zapytań SQL.

Oto kilka kluczowych cech i funkcji biblioteki LINQ:

1. Zapytania LINQ: LINQ wprowadza nową składnię, która umożliwia tworzenie zapytań do różnych źródeł danych. Wyrażenia zapytań LINQ są czytelne i bardziej ekspresywne niż tradycyjne metody programowania.

2. Operacje na kolekcjach: LINQ oferuje bogaty zestaw operacji do manipulacji kolekcjami, takich jak filtrowanie, sortowanie, grupowanie, projekcja, łączenie i wiele innych. Te operacje są dostępne dla różnych typów kolekcji, takich jak List, Array, Dictionary itp.

3. Dostęp do baz danych: LINQ umożliwia wykonywanie zapytań do baz danych, takich jak SQL Server, Oracle, MySQL itp., bezpośrednio w kodzie C#. Używając LINQ to SQL lub Entity Framework, można mapować tabele baz danych na obiekty i wykonywać zapytania przy użyciu składni LINQ.

4. Obsługa danych XML: LINQ umożliwia manipulację danymi XML za pomocą LINQ to XML. Można tworzyć, modyfikować i zapytać dane XML przy użyciu prostych i intuicyjnych wyrażeń LINQ.

Dzięki bibliotece LINQ programiści mogą pisać bardziej czytelny i wydajny kod, redukując ilość pisanego kodu i zwiększając możliwości manipulacji danymi. LINQ jest powszechnie stosowany w aplikacjach .NET, zarówno na platformie desktopowej, jak i w tworzeniu gier w Unity, gdzie można wykorzystać jego funkcje do operacji na kolekcjach i zapytań do baz danych.

Hermetyzacja kodu w Unity

Hermetyzacja kodu w Unity odnosi się do praktyki tworzenia klas i skryptów w taki sposób, aby ukryć szczegóły implementacyjne i chronić dane wewnętrzne przed bezpośrednim dostępem i modyfikacją z innych części kodu. Hermetyzacja jest jednym z podstawowych zasad programowania obiektowego, które promuje izolację i kontrolę dostępu do składowych obiektu.

W Unity hermetyzacja kodu jest szczególnie istotna, ponieważ pozwala na tworzenie modułowych i skalowalnych projektów, zwiększa bezpieczeństwo i ułatwia późniejsze modyfikacje i utrzymanie kodu.

Oto kilka technik hermetyzacji kodu w Unity:

1. Modyfikatory dostępu: Wykorzystaj modyfikatory dostępu takie jak `public`, `private` i `protected`, aby kontrolować dostęp do składowych klasy. Składowe oznaczone jako `private` są dostępne tylko w obrębie danej klasy, `public` umożliwia dostęp z dowolnego miejsca w kodzie, a `protected` pozwala na dostęp tylko z klasy bazowej i klas pochodnych.

2. Właściwości (Properties): Zamiast bezpośredniego dostępu do pól klasy, użyj właściwości (getters i setters) do kontrolowanego dostępu do danych. Możesz ustawić `private set` lub `protected set`, aby umożliwić tylko odczyt lub kontrolowany zapis danych.

3. Metody dostępu: Jeśli chcesz umożliwić dostęp do danych w klasie, ale nadal chcesz kontrolować sposób ich manipulacji, użyj specjalnych metod dostępu, takich jak `Get()` i `Set()`. Możesz w nich zaimplementować dodatkową logikę lub sprawdzanie poprawności danych przed zwróceniem lub zapisem wartości.

4. Ukrywanie szczegółów implementacyjnych: Jeśli masz pewne składowe, które nie powinny być widoczne dla innych klas lub skryptów, możesz je oznaczyć jako `private` lub użyć modyfikatora `internal`, który ogranicza dostęp tylko do komponentów w tym samym projekcie.

5. Interfejsy: Korzystaj z interfejsów, aby zdefiniować zestaw metod i właściwości, które będą dostępne publicznie, jednocześnie ukrywając implementację w klasach. Interfejsy pozwalają na polimorfizm i wymuszają określone zachowanie obiektów.

Hermetyzacja kodu jest ważnym aspektem tworzenia dobrze zorganizowanych i łatwo zarządzalnych projektów w Unity. Poprawna hermetyzacja pozwala na kontrolę dostępu do danych, unikanie bezpośrednich manipulacji, izolację funkcjonalności i tworzenie elastycznych struktur kodu.

String

Oto kilka przykładowych funkcji i metod, które można użyć do operacji na stringach:

1. `Length`: Zwraca liczbę znaków w danym ciągu.
```csharp
string str = "Hello World";
int length = str.Length; // 11
```

2. `ToUpper` i `ToLower`: Zamieniają wszystkie znaki w ciągu na duże lub małe litery.
```csharp
string str = "Hello World";
string upperCase = str.ToUpper(); // "HELLO WORLD"
string lowerCase = str.ToLower(); // "hello world"
```

3. `Substring`: Zwraca podciąg z danego ciągu, począwszy od określonej pozycji.
```csharp
string str = "Hello World";
string subStr = str.Substring(6); // "World"
```

4. `Split`: Dzieli ciąg na podciągi na podstawie określonego separatora i zwraca tablicę wyników.
```csharp
string str = "Apple,Orange,Banana";
string[] fruits = str.Split(','); // ["Apple", "Orange", "Banana"]
```

5. `Concat`: Łączy dwa lub więcej ciągów w jeden.
```csharp
string str1 = "Hello";
string str2 = "World";
string result = string.Concat(str1, " ", str2); // "Hello World"
```

6. `Replace`: Zamienia wszystkie wystąpienia określonego podciągu na inny podciąg.
```csharp
string str = "Hello World";
string replaced = str.Replace("Hello", "Hi"); // "Hi World"
```

7. `Trim`: Usuwa początkowe i końcowe białe znaki z ciągu.
```csharp
string str = "   Hello World   ";
string trimmed = str.Trim(); // "Hello World"
```

8. `Contains`: Sprawdza, czy dany ciąg zawiera określony podciąg.
```csharp
string str = "Hello World";
bool contains = str.Contains("World"); // true
```

9. `IndexOf` i `LastIndexOf`: Znajdują indeks pierwszego lub ostatniego wystąpienia określonego podciągu w ciągu.
```csharp
string str = "Hello World";
int index = str.IndexOf("o"); // 4
int lastIndex = str.LastIndexOf("o"); // 7
```

10. `Parse` i `TryParse`: Konwertują ciąg na wartość liczbową, taką jak int, float, czy double.
```csharp
string str = "123";
int number = int.Parse(str); // 123

string invalidStr = "abc";
bool success = int.TryParse(invalidStr, out int result); // false, result = 0
```

Te to tylko niektóre z wielu funkcji dostępnych do manipulacji i przetwarzania stringów w języku C#. Istnieje wiele innych przydatnych funkcji, które można znaleźć w dokumentacji języka C# i .NET Framework.

Delta time i Time

Delta time i Time są dwoma ważnymi pojęciami w Unity związanymi z kontrolą czasu w grze.

1. Delta Time: Delta time (czas delta) oznacza czas, który upłynął między dwoma kolejnymi klatkami gry. Jest to wartość używana do synchronizacji animacji, ruchu obiektów i innych aspektów gry, aby były płynne niezależnie od prędkości działania samej gry. Delta time jest zazwyczaj używany w połączeniu z innymi wartościami, takimi jak prędkość obiektu, aby obliczyć odległość, jaką powinien pokonać w danym kroku czasowym. W Unity wartość delta time jest dostępna za pomocą `Time.deltaTime`.

```csharp
void Update()
{
    float speed = 5f;
    float distanceToMove = speed * Time.deltaTime;

    transform.Translate(Vector3.forward * distanceToMove);
}
```

W powyższym przykładzie `Time.deltaTime` jest używany do przesunięcia obiektu o równą odległość niezależnie od częstotliwości odświeżania klatek gry.

2. Time: Klasa `Time` w Unity dostarcza różne funkcje i właściwości związane z kontrolą czasu w grze. Pozwala na dostęp do informacji takich jak bieżący czas, czas rozpoczęcia gry, czas trwania gry i inne.

```csharp
void Start()
{
    float startTime = Time.time;
}

void Update()
{
    float elapsedTime = Time.time - startTime;
    Debug.Log("Elapsed Time: " + elapsedTime);
}
```

W tym przykładzie `Time.time` jest używane do odczytania aktualnego czasu w sekundach. W połączeniu z innymi operacjami, takimi jak odejmowanie, można obliczyć czas trwania gry lub różnicę czasową między dwiema operacjami.

Obie klasy `Delta Time` i `Time` są istotne w kontekście kontroli czasu w grze Unity i są powszechnie używane przy tworzeniu animacji, ruchu obiektów, zarządzaniu czasem gry i wielu innych aspektach programowania gier.

W Unity istnieje wiele funkcji i właściwości dostępnych w klasie `Time`, które można stosować do kontroli czasu w grze. Oto kilka przykładów najczęściej używanych funkcji:

1. `Time.deltaTime`: Zwraca czas (w sekundach) między dwoma kolejnymi klatkami gry. Jest to wartość używana do płynnego ruchu obiektów i animacji, niezależnie od prędkości działania gry.

```csharp
void Update()
{
    transform.Translate(Vector3.forward * speed * Time.deltaTime);
}
```

2. `Time.fixedDeltaTime`: Zwraca stałą wartość czasu (w sekundach) między kolejnymi krokami w fizyce gry. Jest to przydatne w przypadku manipulowania fizyką obiektów w grze.

```csharp
void FixedUpdate()
{
    rigidbody.AddForce(Vector3.up * force, ForceMode.Impulse);
}
```

3. `Time.time`: Zwraca bieżący czas (w sekundach) od rozpoczęcia uruchomienia gry. Jest używany do śledzenia czasu trwania gry lub do synchronizacji pewnych akcji w określonych momentach.

```csharp
void Update()
{
    if (Time.time >= nextSpawnTime)
    {
        SpawnObject();
        nextSpawnTime = Time.time + spawnInterval;
    }
}
```

4. `Time.timeScale`: Określa skalę czasu, która kontroluje szybkość odtwarzania gry. Ustawienie wartości na 1 oznacza normalną prędkość gry, wartość 0 zatrzymuje grę, a wartość większa niż 1 przyspiesza ją.

```csharp
void Update()
{
    if (Input.GetKeyDown(KeyCode.P))
    {
        if (Time.timeScale == 0f)
        {
            Time.timeScale = 1f;
        }
        else
        {
            Time.timeScale = 0f;
        }
    }
}
```

5. `Time.smoothDeltaTime`: Zwraca uśredniony czas delta, który jest używany do wygładzania ruchu obiektów. Jest przydatny w przypadku płynnego poruszania się obiektów niezależnie od fluktuacji wydajności.

```csharp
void Update()
{
    transform.Translate(Vector3.right * speed * Time.smoothDeltaTime);
}
```

To tylko kilka przykładów funkcji dostępnych w klasie `Time`. Istnieją także inne, takie jak `Time.realtimeSinceStartup`, `Time.captureDeltaTime`, `Time.captureFramerate` itp., które można używać w zależności od konkretnych potrzeb w projekcie.

Regex

Regex (Regular Expressions) to potężne narzędzie do manipulacji i analizy tekstów. Jest to technika oparta na wzorcach, która umożliwia wyszukiwanie i manipulację tekstowymi danych na podstawie określonych wzorców.

W programowaniu, wyrażenia regularne są często stosowane do:

1. Wyszukiwania: Regex pozwala na odnalezienie konkretnych wzorców w tekście. Można używać go do sprawdzania, czy dany tekst pasuje do określonego wzorca, jak np. sprawdzanie poprawności formatu adresu e-mail.

```csharp
using System;
using System.Text.RegularExpressions;

public class Program
{
    public static void Main()
    {
        string email = "example@example.com";
        string pattern = @"^[a-zA-Z0-9]+@[a-zA-Z0-9]+\.[a-zA-Z]{2,}$";

        bool isMatch = Regex.IsMatch(email, pattern);

        if (isMatch)
        {
            Console.WriteLine("Adres e-mail jest poprawny.");
        }
        else
        {
            Console.WriteLine("Nieprawidłowy adres e-mail.");
        }
    }
}
```

2. Podstawiania: Regex umożliwia zastępowanie określonych wzorców w tekście innymi wartościami. Można go używać do modyfikacji i formatowania tekstów.

```csharp
using System;
using System.Text.RegularExpressions;

public class Program
{
    public static void Main()
    {
        string input = "Hello, [Name]! Today is [Day].";
        string pattern = @"\[([A-Za-z]+)\]";

        string result = Regex.Replace(input, pattern, (match) =>
```

```
        {
            string placeholder = match.Groups[1].Value;
            if (placeholder == "Name")
            {
                return "John Doe";
            }
            else if (placeholder == "Day")
            {
                return DateTime.Now.DayOfWeek.ToString();
            }
            return match.Value;
        });

        Console.WriteLine(result);
    }
}
```

3. Walidacji: Regex może być używany do sprawdzania poprawności formatu danych, jak np. numeru telefonu, kodu pocztowego, identyfikatora itp.

```csharp
using System;
using System.Text.RegularExpressions;

public class Program
{
    public static void Main()
    {
        string phoneNumber = "123-456-7890";
        string pattern = @"^\d{3}-\d{3}-\d{4}$";

        bool isValid = Regex.IsMatch(phoneNumber, pattern);

        if (isValid)
        {
            Console.WriteLine("Numer telefonu jest poprawny.");
        }
        else
        {
            Console.WriteLine("Nieprawidłowy numer telefonu.");
        }
    }
}
```

Regex oferuje szeroki zakres funkcji i składni, które umożliwiają bardziej zaawansowane operacje na tekstach, takie jak grupowanie, kwantyfikatory, negacje, operatory logiczne i wiele innych. Może być stosowany w różnych językach programowania, w tym także w Unity.

Przydatne link

JSON Formatter & Validator (curiousconcept.com)
Free Online JSON Escape / Unescape Tool - FreeFormatter.com
Unity - Manual: Unity User Manual 2021.3 (LTS) (unity3d.com)
Unity - Scripting API: (unity3d.com)
regex101: build, test, and debug regex
C# Programming Guide | Microsoft Learn
Base64 to Image | Decode and Encode Online (base64-to-image.com)