Mateusz Wiliński

GAMEDEV

PROGRAMOWANIE C#
W UNITY

--2023.11

GAMEDEV

PROGRAMOWANIE C#
W UNITY

Autor: Mateusz Wiliński

Email: info@astraja.com

https://astraja.com/

Wprowadzenie

Ebook Gamedev Programowanie C# w Unity przeznaczony jest dla osób rozpoczynających naukę programowania w języku C#, chcących tworzyć gry komputerowe w silniku Unity.

Unity udostępnia dodatkowe klasy oraz wykorzystuje specyficzny sposób programowania. Publikacja skupia się głównie na zastosowaniu języka C# w silniku Unity.

Link do dysku google z Unity Assets:
https://drive.google.com/drive/folders/1Mmg-KUS9GM4n0b-2kRlUrLFha2DSY8sB?usp=sharing

Wszelkie uwagi, chęć pomocy, informacje o błędach prosimy kierować na adres:
astraja.pro@gmail.com

Code color theme (VSC Light +)

Spis treści

Podstawowe skróty Unity

W	Move tool
T	Rect tool
F	Focus on selected object
V	Vertex Snapping
Ctrl + D	Duplicate
Alt + LMB	Orbit view around selection
Ctrl + F	Find Object
Ctrl + P	Play

Podstawowe skróty Visual Studio

Ctrl + D	Duplicate
Ctrl + R, Ctrl + R	Rename
Ctrl + .	Quick Action and Refactoring
Ctrl + Shift + Space	Check Overloads (cursor inside parenthesis)
Alt + Up / Down	Move line / lines up or down
Shift + Alt + Down	Line down extend column
Ctrl + L	Cut line
Ctrl + K, Ctrl + C	Comment section
Shift + F12	Find all references
F12	Go To Definition

Skrypty w Unity

Zasady tworzenia skryptów:

1. Skrypt definiuje nową klasę (a tym samym, nowy Component)
2. Nazwę klasy tworzy się zgodnie z notacją PascalCase, tj. Pierwszy znak każdego słowa składowego piszę się z wielkiej litery, np. PlayerMovement, Hp, Enemy, EnemyShooting
3. Należy używać tylko liter alfabetu angielskiego (bez polskich znaków)
4. Nie wolno używać spacji
5. Nie wolno rozpocząć nazwy od cyfry
6. Nazwy powinny być deskryptywne (opisowe)

Skrypty tworzone są na podstawie szablonu, który można zmienić i dopasować do własnych potrzeb. W zainstalowanej wersji edytora należy znaleźć plik z szablonem 81-C# Script-NewBehaviourScript.cs i go edytować (najlepiej odpowiednim programem (Visual Studio Code), aby zapisać go z uprawnieniami administratora). Ścieżka dostępu wygląda następująco:

C:\Program Files\Unity\Hub\Editor\2022.2.15f1\Editor\Data\Resources\ScriptTemplates

Klasa

```
public class DebugowanieWKonsoli : MonoBehaviour
{
    // zawartość klasy
}
```

Klasa (class) jest instrukcją tworzenia obiektu w grze.

Wewnątrz nawiasów klamrowych podaje się pola, właściwości i metody, które będą opisywać tworzony na jej podstawie obiekt.

Wewnątrz klasy Gracz, można zdefiniować takie pola jak szybkość, atak, obrona, poziom, itp.

Klasa może zawierać metody, czyli zachowania gracza, np. Atakuj, Skacz, PoruszajSie, itp.

Dwukropek (:) wstawiony po nazwie klasy, oznacza, że dana klasa dziedziczy z innej klasy. W Unity większość klas dziedziczy z bazowej klasy MonoBehaviour, dzięki czemu ma dostęp do wszystkich bibliotek.

Dana klasa może dziedziczyć tylko z jednej klasy.

Właściwości i metody, dostępne w danej klasie, widoczne są po wpisaniu nazwy klasy i dodaniu znaku kropki (.)

Namespace

```
using UnityEngine;
```

Namespace (inaczej przestrzenie nazw / biblioteki) to kontenery w których znajdują się klasy, pogrupowane w odpowiedni sposób.

Dzięki instrukcji using, uzyskujemy łatwy dostęp do klas i funkcjonalności w ramach danej biblioteki.

Instrukcja `using UnityEngine` pozwala korzystać z podstawowych funkcji stworzonych specjalnie dla silnika Unity.

Komentarze

Komentarze stosuje się przede wszystkim w dwóch przypadkach:

1. do objaśnienia bloku kodu
2. do dezaktywacji bloku kodu, by chwilowo nie był wykonywany

Występują 2 typy komentarzy:

```
// Komentarz liniowy

/* Komentarz     blokowy,
wieloliniowy
 */
```

Debugowanie informacji w konsoli

Metoda `Debug.Log()` służy do szybkiego wyświetlania w konsoli informacji związanych z grą w środowisku produkcyjnym.

Wewnątrz nawiasu, jako argument można podać wiele typów informacji, które mają być wyświetlone.

Przykłady:

Wyświetlenie wyniku działania matematycznego:

```
Debug.Log(12 * 45);
```

Wyświetlenie nazwy obiektu do którego dołączony jest skrypt

```
Debug.Log(name);
```

Wyświetlanie pozycji obiektu na scenie `Debug.Log(transform.position);`

Kolejność eventów w pętli Unity

W Unity występują 2 pętle, które zawierają w sobie zdefiniowane metody / eventy. Metody te wywołują się automatycznie w określonej przez silnik Unity kolejności.

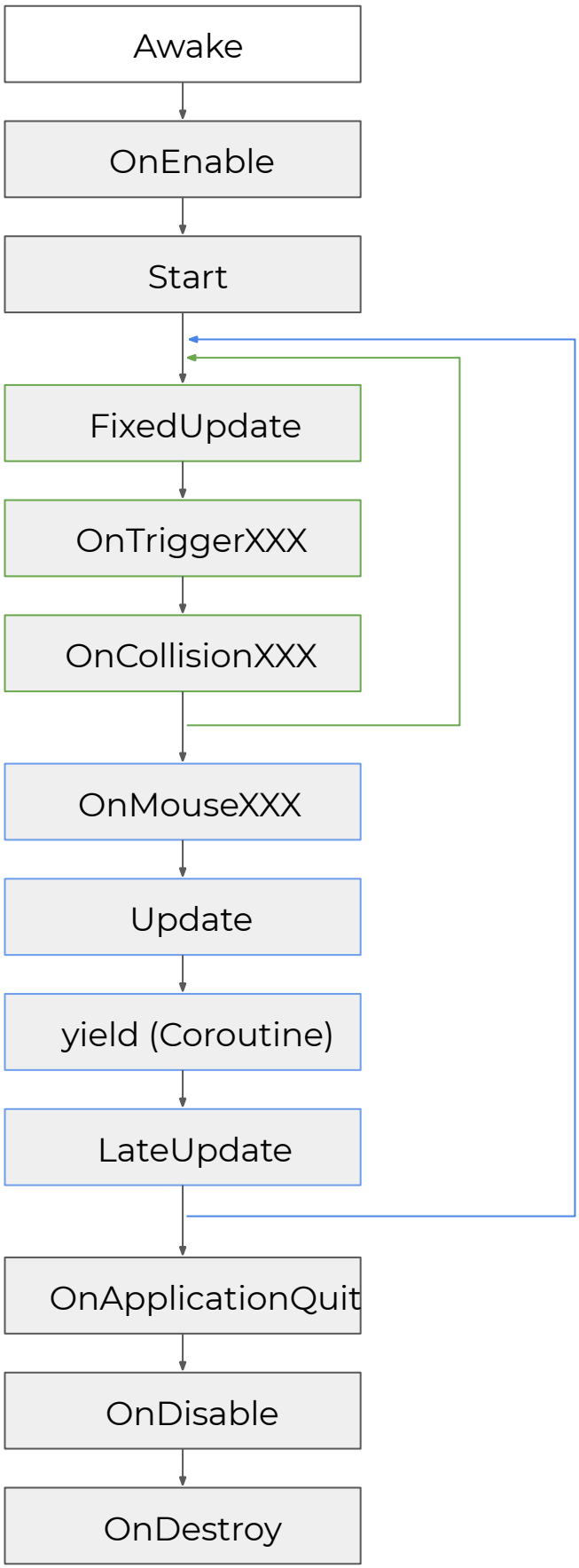
Wszystkie dostępne elementy pętli znajdują się w dokumentacji Unity: https://docs.unity3d.com/Manual/ExecutionOrder.html a ich uproszczona wersja na kolejnej stronie.

Elementy pętli wykonywane są w każdej klatce dla każdego aktywnego obiektu na scenie. W przypadku, gdy dana metoda jest pusta, należy ją usunąć ze skryptu.

Pętla Update odpowiada za interakcje z użytkownikiem, instrukcje, które muszą wykonywać się nieustannie oraz renderowanie obrazu. Dla każdego komputera pętla Update będzie wykonywać się odmiennie, w zależności od mocy i złożoności gry. Pętla Update decyduje o ilości fps w gotowej grze.

Pętla FixedUpdate odpowiada za obliczenia związane z fizyką w grze i jest wykonywana w stałych odstępach czasu, tj. 50 razy na sekundę.

Kolejność eventów w pętli Unity



https://docs.unity3d.com/Manual/ExecutionOrder.html

Awake()

Metoda Awake wykona się 1 raz, jako pierwsza, w momencie utworzenia obiektu w grze.

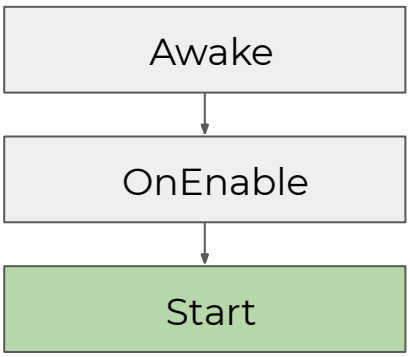
```
void Awake()
{
    Debug.Log("Awake!");
}
```

Metoda Awake służy głównie do inicjowania danego obiektu i zapisywania referencji do komponentów tego obiektu w zmiennych.

Cykle wykonywania metod w skrypcie można prześledzić tutaj:

https://docs.unity3d.com/Manual/ExecutionOrder.html

Start()



Metoda Start() wykona się 1 raz, tuż po tym, jak wszystkie metody

Awake i OnEnable zakończą swoje działanie.

```
void Start()
{
    Debug.Log("Start!");
}
```

W metodzie Start można m.in. tworzyć powiązania z innymi obiektami na Scenie.

Należy pamiętać, że nigdy nie można przewidzieć w którym obiekcie na Scenie metoda Start wykona się jako pierwsza.

Cykle wykonywania metod w skrypcie można prześledzić tutaj:

https://docs.unity3d.com/Manual/ExecutionOrder.html

Update()

Metoda Update() wykonuje się ciągle, wiele razy w trakcie każdej sekundy. Ilość wywołań jest zmienna, zależy od mocy komputera i złożoności wyświetlanej sceny.

```
void Update()
{
    Debug.Log(Time.time);
}
```

Metoda Update() zacznie wykonywać się automatycznie tuż po tym, jak dany skrypt (lub obiekt z tym skryptem) zostanie załadowany i aktywowany na scenie oraz po tym jak funkcja Start() zakończy swoje działanie.

Metoda Time.time pokazuje czas, który upłynął od chwili uruchomienia gry

Cykle wykonywania metod w skrypcie można prześledzić tutaj:

https://docs.unity3d.com/Manual/ExecutionOrder.html

LateUpdate()

Metoda LateUpdate() działa podobnie jak Update, ale wykonuje się odrobinę później (dlatego late ;)

Używa jej się, by wykonać operacje, które bazują na informacjach przetworzonych w metodzie Update.

Przykładem może być kamera, która ma podążać za graczem. Ruch gracza może być wykonywany w funkcji Update, dlatego ruch kamery należy wykonać odrobinę później, czyli w funkcji LateUpdate, kiedy ruch gracza zostanie zakończony.

Cykle wykonywania metod w skrypcie można prześledzić tutaj:

https://docs.unity3d.com/Manual/ExecutionOrder.html

FixedUpdate()

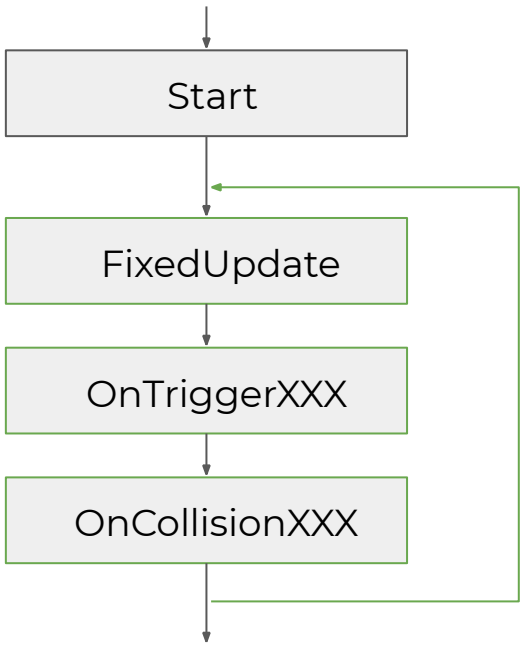
Metoda FixedUpdate, w odróżnieniu od Update / LateUpdate, wykonuje się określoną przez system ilość razy w ciągu sekundy.

Domyślnie jest to wartość 50 klatek na sekundę.

Wewnątrz metody FixedUpdate zazwyczaj zawiera się logikę gry związaną z obliczeniami fizyki.

Będą tam tworzone promienie (RayCast), zachowania związane z komponentem RigidBody czy zderzenia.

FixedUpdate wraz z eventami odpowiedzialnymi za wykrywanie kolizji wywołują się w niezależnej pętli.



Time.deltaTime

W związku z tym, że metoda Update() wykonuje się w różnych odstępach czasu, należy to skompensować, aby na każdym komputerze rozgrywka wyglądała podobnie. W tym celu korzysta się z właściwości Time.deltaTime, która zwraca czas jaki upłynął od wyświetlenia ostatniej klatki. Operacje wykonywane w metodzie Update mnoży się przez tę liczbę, dzięki czemu każdy gracz uzyska podobne doświadczenia z gry, np:

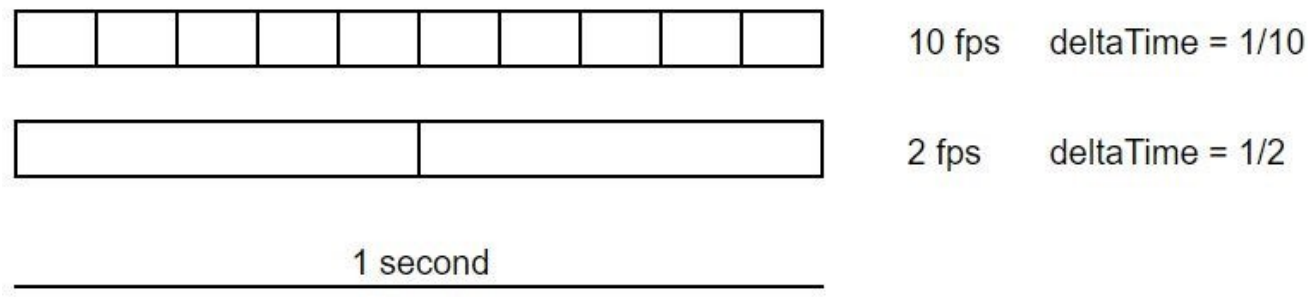
```
transform.Translate(new Vector3(1f, 0f, 0f) * Time.deltaTime);
```

Jak to działa?

Przykład dwóch komputerów: jeden działa z prędkością 10fps, a drugi 2 fps.

Na pierwszym komputerze metoda Update wykona się 10 razy w ciągu sekundy, a na drugim tylko 2 razy. Chcąc wykonać ruch, normalnie, na pierwszym komputerze dany obiekt poruszałby się 5 x szybciej.

Dzięki **Time.deltaTime** można to wyrównać.



Mnożąc wartości przez deltaTime, otrzymuje się mniejsze wyniki na jedną klatkę dla szybkich komputerów i większe wyniki dla wolniejszych komputerów, dzięki czemu ostatecznie rozgrywka staje się wyrównana.

Typy danych (data types)

Język C# rozróżnia wiele typów danych, np.: liczby całkowite, liczby ułamkowe, znaki, tekst, wartości logiczne i setki innych.

Każda nowo utworzona klasa(skrypt) oraz każdy komponent jest osobnym typem danych.

Data type	Example	Size
int	-10, 0, 128, 2056	4 bytes
float	-12.5f, 0.01f, 2023.24f	4 bytes
bool	true, false	1 byte
byte	0, 99, 255	1 byte
Vector3 (struct)	new Vector3 (3f, 0f, 0f)	3 x 4 bytes
Color32 (struct)	new Color32 (255, 0, 0, 255)	4 x 4 bytes
string	"Happy Zombie Games"	2 bytes * liczba znaków (ref)
Transform	Object	ref
GameObject	Object	ref

Var

W języku C# istnieje specjalny typ: **var**, który może zastępować dowolny inny typ.

W momencie kompilacji, kompilator sprawdzi jakiego typu jest podana wartość i zastosuje odpowiedni typ.

Zmienne typu var muszą być zadeklarowane wewnątrz funkcji.

Zmienne (variables)

Zmienna to wirtualny kontener, który może przechować informacje danego typu. Podczas deklaracji zmiennej podaje się :

- typ danych jaki będzie mógł być w niej zapisany,
- unikalną nazwę, za pomocą której dana zmienna będzie wywoływana.

Deklaracja zmiennej:

```
var age;
```

W dalszej części programu, za pomocą nazwy zmiennej, można odczytywać, przypisywać i nadpisywać wartości tych zmiennych za pomocą ich nazw, np.:

```
age = 12;

Debug.Log(age);
```

Możliwe jest przypisanie wartości początkowych do zmiennych w czasie ich tworzenia, co nazywane jest inicjalizacją, np:

```
var distance = 1267865421;
```

Nazewnictwo zmiennych

Zasady tworzenia zmiennych:

1. Nazwy zmiennych należy tworzyć według konwencji **camelCase**, tzn: nazwa zaczyna się od małej litery, a każde kolejne słowo składowe zaczyna się z wielkiej litery, np: bodyTemperature, houseNumber, nextPosition
2. Nazwy powinny być deskryptywne, aby każdy mógł się domyślić ich zawartości
3. W nazwie można używać cyfr, ale nigdy na początku
4. W nazwie nie używa się spacji
5. Nie należy stosować polskich liter

Wszelkie nazewnictwo jak i cały kod, powinien być pisany w języku angielskim

Zmienne lokalne / globalne

Zmienną można tworzyć bezpośrednio wewnątrz klasy, nazywamy ją wtedy polem lub zmienną globalną. Zmienna taka jest dostępna wewnątrz całego skryptu.

```
public class Player : MonoBehaviour
{
    private float speed = 5.0f;
}
```

Zmienną globalną można inicjalizować podczas tworzenia. W przypadku samej deklaracji takiej zmiennej, przypisanie lub nadpisanie wartości musi nastąpić wewnątrz funkcji.

Zmienną lokalną tworzy się wewnątrz bloku kodu, np. funkcji, instrukcji warunkowej. Zmienna taka istnieje tylko i wyłącznie w tym zakresie, nie można się do niej odwołać z zewnątrz.

```
public class Player : MonoBehaviour
{
      private void Awake()
      {
            private float speed = 5.0f;
      }
}
```

Modyfikatory dostępu

W języku C# istnieje kilka modyfikatorów dostępu, które pozwalają ukryć lub uwidocznić zawartość skryptu (zmienne, funkcje i inne elementy).

Podawanie modyfikatorów dostępu jest opcjonalne. Domyślnie, jeśli żaden nie zostanie podany, to tworzona zmienna czy funkcja jest prywatna i będzie dostępna tylko i wyłącznie wewnątrz danego skryptu.

Modyfikatory dostępu podaje się przed zmienną, funkcją, klasą, itp.

```
public void AddPoint()
{
    // instruction to increase points number
}
```

Wszystkie zmienne powinny być prywatne, a tylko te funkcje, które mają być dostępne z zewnątrz, powinny być publiczne (lub opatrzone innym modyfikatorem ze względu na sytuację)

Dużym ułatwieniem jest wyświetlanie zmiennych w inspektorze, gdzie można dołączać referencje do innych obiektów. Jest możliwe po dodaniu modyfikatora public, ale łamie to powyższą zasadę enkapsulacji (hermetyzacji).

Unity stworzyło specjalny atrybut **[SerializeField]**, który pozwala zachować zmienną prywatną, a jednocześnie wyświetlić ją w Inspektorze.

Atrybut [SerializeField] można dodać w jednej linii lub nad zmienną.

```
[SerializeField] GameObject prefab;

[SerializeField] private float speed;
```

Modifier	Description
private	The code is only accessible within the same class. By default, all members of a class are private if you don't specify an access modifier.
public	The code is accessible for all classes.
protected	The code is accessible within the same class, or in a class that is inherited from that class.
internal	The code is only accessible within its own assembly, but not from another assembly.
[SerializeField]	This atribute is very common in Unity. It keeps elment private but shows it in Inspector so that refereneces can be made

Liczby całkowite - int

Podstawowy typ prosty, reprezentujący liczby całkowite to integer (int).

Zajmuje on 4 bajty pamięci (32bity) i może przechować liczby w zakresie od -2 147 483 648 do 2 147 483 647. Zakres taki oblicza się w następujący sposób:

Komputery działają w systemie dwójkowym, mogą przyjmować wartość 0 lub 1. Każdy bit pamięci może przyjąć jedną z dwóch wartości, a liczba zapisywana jest na 32 bitach.

Maksymalną wartość oblicza się podnosząc liczbę 2 do potęgi 32.

2^32 = 4,294,967,296

Typ int obsługuje liczby ujemne i dodatnie, dlatego powyższy wynik dzielimy przez 2(w uproszczeniu), co daje nam zakres liczb, które mogą być zapisane za pomocą tego typu. Należy uważać, by nie przekroczyć tego zakresu!

Przykład utworzenia zmiennej o typie int:

```
int points = 12;
```

Innym typem, który przechowuje tylko liczby dodatnie od 0 do 255 jest **byte**. Używa się go bardzo często do zapisu wartości składowych kanałów koloru.

Liczby ułamkowe - float

Typ liczb zmiennoprzecinkowych float zajmuje 32 bity pamięci (4 bajty), a jego precyzja wynosi 7 cyfr.

Do liczb typu float dodaje się przyrostek **f**, np: 0.01f, 12.1f, -0.0001f

Miejsca ułamkowe oddziela się od całości za pomocą kropki.

Wartości typu float mogą przechowywać bardzo duże liczby, ale z małą precyzją. Co to oznacza?

Np. liczba 72 678 942 będzie zaokrąglona do 7 cyfr znaczących, tj. 72

678 940 i zapisana w postaci wykładniczej jako: 7.267894E+07

Zapis 7.267894E+07 oznacza, że liczbę 7.267894 mnożymy przez 10^7 czyli przesuwamy przecinek o 7 pozycji w prawo, co daje liczbę: 72 678 940

Wartości typu float które zawierają więcej niż 7 cyfr znaczących zostaną zaokrąglone.

Przykładowo liczba 23 879 234 657 zostanie zapisana za pomocą typu float jako 2.387924E+10, czyli 23 879 240 000.

Przykład utworzenia zmiennej o typie float:

```
float distance = 0.05f;
```

W większości przypadków większa precyzja nie jest wymagana, ale jeśli zajdzie taka potrzeba należy użyć typu double lub decimal.

Wartości logiczne - bool

Wartości logiczne boolean (bool) mogą przyjmować wartość **true** lub **false**.

Na ich podstawie działają komputery, które zbudowane są z milionów tranzystorów, które przyjmują dwa stany: 1 i 0, czyli prawda i fałsz. Cały system opiera się na przełączaniu tych dwóch stanów na milionach mikroskopowych przekaźników.

Na podstawie wartości bool możemy np. sprawdzać:

czy gracz dotknął ziemi, czy klawisz myszy został naciśnięty, czy menu zostało aktywowane,

Przykłady tworzenia zmiennych typu bool:

```
bool isGrounded = true;
bool isMenuOpen = false;
```

W Unity istnieje wiele metod, które sprawdzają zachowania w grze i zwracają wartości typu bool. Często takie metody przypisuje się do zmiennych lub używa się ich w instrukcjach warunkowych.

Vector3 / Color32

W Unity występuje wiele typów, które składają się z wartości innych typów prostych. Przykładem może być typ Vector2 lub Vector3, który jest strukturą złożoną z 2 lub 3 osobnych wartości typu float.

Typy takie tworzy się z użyciem słowa kluczowego new, jak poniżej:

```
Vector2 pos2D = new Vector2 ( 0.1f, 0f);
Vector3 pos3D = new Vector3 ( 12.2f, 0f, 0f);
```

Innym przykładem jest zmienna typu Color32. Struktura ta składa się z 4 wartości typu byte, które określają poszczególne wartości kanałów koloru od 0 do 255 (red, green, blue, opacity).

Przykład poniżej jest reprezentacją koloru czerwonego o pełnym kryciu (brak przezroczystości):

```
Color32 color = new Color32 (255, 0, 0, 255);
```

Kolory w Unity można również przedstawiać za pomocą typu Color, który przyjmuje wartości ułamkowe od 0.0 do 1.0, ale jest to rozwiązanie mniej optymalne.

```
Color color = new Color (0.00f, 0.01f, 0f, 1.0f);
```

Wartości tekstowe - string

Wartości tekstowe w programowaniu zapisywane są za pomocą typu string. Dane takie powinny być zawsze zawarte wewnątrz cudzysłowu, np.: `"John"`

Typ string różni się od typów prostych takich jak: int, float, bool.

Wielkość pamięci zajmowanej przez zmienną typu string zależy od jej zawartości, czyli ilości znaków. Pod uwagę należy wziąść również sposób kodowania znaków, ale można przyjąć, że każdy znak w ciągu zajmuje 2 bajty pamięci. Dodatkowo 2 bajty są przeznaczone na określenie długości wartości tekstowej. Ciąg znaków `"Angelika",` może zajmować w pamięci 8 x 2 + 2 = 18 bajtów.

Przykład utworzenia zmiennej typu string:

```
string name = "Azgaran";
```

Typ string jest niezmienny (immutable), to znaczy, że nie można zmienić jego zawartości. W chwili pozornego nadpisywania zmiennej typu string, w pamięci komputera, na stercie (heap), tworzony jest nowy obiekt. W zmiennej (na stosie) przechowywany jest tylko adres do danej komórki pamięci, a nie zapisana wartość. Poprzednia wartość zostanie z czasem usunięta, z pamięci, za pomocą tak zwanego GB (Garbage Collector). Jest to automatyczne narzędzie, które czyści pamięć z niepotrzebnych informacji.

Częste nadpisywanie zmiennych typu string jest z zasady niekorzystne i niewydajne. W takim przypadku należy skorzystać z narzędzia StringBuilder

```
string myText = "";
myText = "Hello";
myText += "!";
```

Konkatenacja

Konkatenacja to połączenie wartości tekstowych z innymi informacjami.

Istnieje wiele sposobów na połączenie danych, poniżej zostaną pokazane 2 najczęściej stosowane. Należy pamiętać, że konkatenacja ostatecznie tworzy zawsze informacje typu tekstowego.

```
int counter = 0;
```

Konkatenacja z operatorem +

```
Debug.Log("Counter" + counter);
```

Konkatenacja przez interpolację

```
Debug.Log($"Counter {counter}");
```

Obiekty (GameObject, Components)

Wszystkie dane zapisywane w Unity są obiektami, nawet typy proste.

Obiekty, poza typami prostymi (liczbowymi) i stringami, tworzy się za pomocą słowa kluczowego `new`.

W Unity, często przypisuje się wartości w Inspektorze lub za pomocą specjalnych metod, dlatego słowa kluczowego new używa się sporadycznie.

W Unity wszystkie elementy dostępne w grze, są typu **GameObject.**

```
GameObject player;
GameObject weapon;
GameObject bullet;
```

Inkrementacja / dekrementacja

Inkrementacja / dekrementacja to zwiększenie / zmniejszenie zmiennej o daną wartość. Istnieją 3 sposoby zastosowania inkrementacji / dekrementacji:

```
int counter = 0;
counter = counter + 1;
counter += 1; counter++;
```

Najkrótszą wersję ++ / -- stosuje się tylko do zmiany wartości o 1.

W przypadku zmiany o wartość inną niż 1 lub za pomocą innej operacji niż +/-, najczęściej stosuje się formę skróconą, czyli, np: counter += 3;

Przykłady operacji inkrementacji:

```
int number = 0;
number += 5;    // 0 + 5    (5)
number -= 2;    // 5 - 2    (3)
number *= 4;    // 3 * 4    (12)
number /= 2;    // 12 / 2   (6)
number %= 5;    // 6 % 5    (1)
```

Istnieją 2 rodzaje inkrementacji i dekrementacji, gdzie operator ++ lub -- występuje po zmiennej lub przed nią.

```
int counter = 0;
counter++;  //postinkrementacja
++counter;  // preinkrementacja
```

Wykonując instrukcje w osobnej linii, nie ma znaczenia która z nich zostanie użyta.

Ma to jednak znaczenie w przypadku łączenia operacji w jednej linii.

W przykładzie poniżej, najpierw zostanie wyświetlona wartość zmiennej, a dopiero później zostanie ona zwiększona o 1:

```
int counter = 0;

Debug.Log(counter++);
```

W przykładach poniżej zmienna zostanie zwiększona i wyświetlona:

```
Debug.Log(++counter);

Debug.Log(counter+=1);
```

Losowa liczba

Rozgrywka byłaby nudna, gdyby zawsze wyglądała w dokładnie taki sam sposób. Za pomocą losowości można uatrakcyjnić tworzoną grę. Mimo, że język C# posiada klasę Random, która znajduje się w przestrzeni nazw – System, to jednak w Unity, częściej korzysta się z dedykowanej klasy Random.

Możliwe jest tworzenie losowych liczb typu int i float za pomocą metody Range()

```
int randomInt = Random.Range(minInt, maxInt);
float randomFloat = Random.Range(minFloat, maxFloat);
```

W przypadku typu integer, wartość minimalna jest inclusive, a maksymalna exclusive. Oznacza to, że wartość maksymalna nigdy nie zostanie wylosowana. Chcąc ustalić zakres losowania, należy wartość maksymalną zwiększyć o 1.

W przypadku typu float, obie wartości są inclusive.

```
int rand = Random.Range(0, 11); //Losuje liczbę od 0 do 10
float rand2 = Random.Range(0f, 1.0f); // Losuje liczbę od 0 do 1.0
```

Funkcje

Funkcja to blok kodu, który wykonuje określone zadanie. Funkcje wbudowane w biblioteki języka nazywane są metodami. Każda funkcja w języku C# musi być zawarta wewnątrz klasy, dlatego funkcje są tym samym co metody.

Syntax funkcji:

```
typZwrotny MyFunction()
{
    // ciało funkcji (instrukcje do wykonania)
}
```

Powyższa definicja funkcji jest tylko instrukcją, aby ją wykonać należy ją wywołać w odpowiednim miejscu skryptu.

Wywołanie funkcji następuję poprzez podane jej nazwy i nawiasów okrągłych:

```
MyFunction();
```

Funkcja może przyjąć różne typy zwrotne (void, int, float, GameObject, itp). Funkcje, które nie muszą niczego zwracać, a jedynie wykonać pewne instrukcje, przyjmują typ zwrotny **void**.

```
void MyFunction()
{
    // ciało funkcji (instrukcje do wykonania)
}
```

Funkcje z parametrem

Tworzenie funkcji z parametrami pozwala na jej uniwersalność. W zależności od podanych argumentów, wynik wywołania funkcji będzie inny.

Przykład funkcji z parametrami, która oblicza ich sumę:

```
void Calculate(int a, int b)
{
    Debug.Log(a+b);
}
```

Wywołanie funkcji wyświetli wynik zależny od podanych argumentów:

```
Calculate(2, 4);  // wyświetli 6

Calculate(5, 10); // wyświetli 15
```

Przykład funkcji, która wyświetli podaną informację. W grze można to później dostosować tak, by informacja pokazywała się w panelu na ekranie.

```
void DisplayMessage(string message)
{
    Debug.Log(message);
}
DisplayMessage("Level Up");
DisplayMessage("Attack +1");
```

Funkcje z return

Funkcje muszą zawierać instrukcję **return**, jeśli zwracają jakąś wartość, czyli zawsze, kiedy typZwrotny jest inny niż **void**.

Przykład funkcji obliczającej sumę i zwracającej wynik. Zazwyczaj wywołanie takiej funkcji przypisuje się do zmiennej lub używa jako argument innej funkcji:

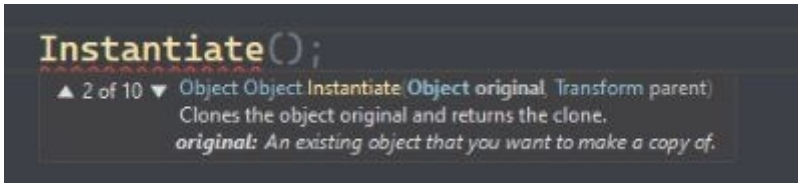
```
int Calculate(int a, int b)
{
      return a + b;
}

int result = Calculate(20, 3);
```

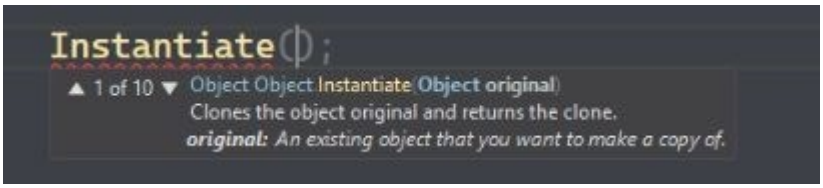
Instrukcji **return** używa się również do przerwania działania funkcji. Kompilator po wykonaniu linii z return, przerwie wykonywanie dalszych instrukcji i opuści daną funkcję.

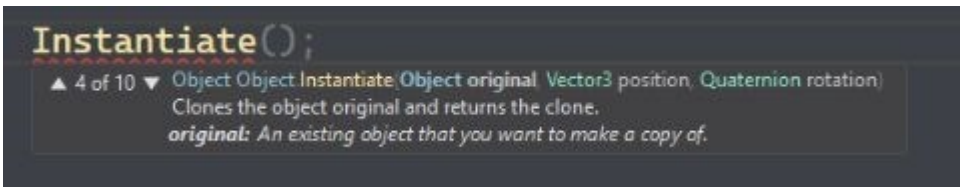
Przeciążanie funkcji

Funkcje mogą posiadać tę samą nazwę, ale inną sygnaturę, czyli inną ilość parametrów lub ich odmienny typ.

W Unity większość metod ma swoje przeciążenia (overloads), np:







Dzięki przeciążaniu, można podać określone argumenty, po to, by uzyskać różne efekty wywołania funkcji. Metoda Instantiate, która spawnuje obiekty na scenie, ma aż 10 różnych konfiguracji (przeciążeń).

Wszystkie przeciążenia można sprawdzić umieszczając kursor wewnątrz nawiasu okrągłego i stosując kombinację klawiszy **Shift + Ctrl + Space**

Przykład przeciążania funkcji

```
int Calculate(int a, int b)
{
      return a + b;
}

float Calculate(float a, float b)
{
      return a + b;
}

int result = Calculate(20, 3);
float resultFloat = Calculate(0.02f, 0.03f);
```

Instrukcja warunkowa if

Instrukcja warunkowa pozwala dokonywać wyborów na podstawie podanych warunków.

Syntax instrukcji warunkowej to:

```
if (condition == true)
{
    // block to be executed if the condition is true
    // or skipped if false
}
```

Instrukcja warunkowa if wykona zadania umieszczone wewnątrz nawiasów klamrowych tylko i wyłącznie wtedy, jeśli podane warunki wewnątrz nawiasu okrągłego będą prawdziwe (condition).

W przykładzie poniżej widać, że funkcja strzelania wywoła się tylko i wyłącznie wtedy, jeśli gracz będzie miał naboje (jeśli ich ilość będzie większa niż 0).

```
int bulletCount = 12;
if (bulletCount > 0)
{
    Shoot();
}
```

Instrukcja warunkowa może składać się z wielu bloków, które będą określać różne warunki. Często instrukcje warunkowe posiadają dodatkowy blok else, który wykona się zawsze wtedy, kiedy żaden warunek nie okazał się prawdziwy. Działa to jak zawór bezpieczeństwa. W instrukcji warunkowej tylko jeden blok może być wywołany, pierwszy, którego warunki były prawdziwe.

Syntax rozbudowanej intrukcji warunkowej:

```
if (condition1)
{
    // instructions to execute if condition1 is true
}
else if (condition2)
{
    // instructions to execute if condition2 is true
}
else
{
    // instructions to execute if none of above is true
}
```

Kompilator czyta kod od góry w dół. Pierwszy prawdziwy warunek wywoła zawarte w danym bloku instrukcje, a reszta bloków zostanie pominięta.

Input GetAxis

Klasa Input zawiera metody i właściwości, dzięki którym można pobierać dane od użytkownika (mysz, klawiatura, pad, dotyk).

Za pomocą Input.GetAxis / Input.GetAxisRaw zwracane są wartości od -1.0 do 1.0 w zależności od wciśniętego klawisza osi (WSAD, strzałki lub joystick na padzie).

Przykład prostej funkcjonalności ruchu dla obiektu 2D

```
void Update()
{
      float speed = 12f;
      float x = Input.GetAxisRaw("Horizontal");
      float y = Input.GetAxisRaw("Vertical");
      transform.Translate(new Vector3(x, y, 0f).normalized
      * speed * Time.deltaTime);
}
```

Input.GetMouseButtonDown()

Input. GetMouseButtonDown() zwraca wartość true, w tej klatce w której zostanie wykryte naciśnięcie przycisku myszy. Podany argument określa przycisk myszy: 0 - LMB, 1 - RMB, 2 - MMB

Przykład wywołań funkcji po wciśnięciu lewego i prawego przycisku myszy:

```
int bulletCount = 12;
void Update()
  {
     if (Input.GetMouseButtonDown(0))
     {
          Shoot();
     }
     if(Input.GetMouseButtonDown(1))
     {
          Reload();
     }
  }

void Shoot()
  {
        bulletCount--;
        Debug.Log($"shoot: {bulletCount}");
}

void Reload()
  {
        bulletCount = 12;
        Debug.Log("reload");
}
```

Operatory porównania

Warunki w instrukcji warunkowej zawsze muszą przyjąć wartość typu bool, true lub false.

Zazwyczaj korzysta się w nich z operatorów porównania (conditional operators):

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Operatory logiczne

Warunki w instrukcjach warunkowych można łączyć za pomocą specjalnych warunkowych operatorów logicznych (logical operators). Czasami dane zdarzenie może nastąpić dopiero po określeniu wielu warunków, co może wydawać się dość skomplikowane. Działania związane z wartościami logicznymi (true i false) są na tyle istotne w programowaniu, że istnieje osobna dziedzina - Algebra Boola, która pomaga zrozumieć skomplikowane operacje logiczne i sposoby ich uproszczenia. W Unity, w większości przypadków, możemy się obejść bez tego ;)

Poniżej tabela z operatorami logicznymi: AND(koniunkcja), OR (alternatywa) i NOT (negacja).

Operator	Name	Example
&&	and	Returns True if both statements are true
||	or	Returns True if one of the statements is true
!	not	Reverse the result, returns False if the result is true

Operator &&

Przykład strzału po wciśnięciu LMB i przy liczbie naboi powyżej 0

```
if (Input.GetMouseButtonDown(0) && bulletCount > 0)
{
    Shoot();
}
```

Operator ||

Przykład strzału po wciśnięciu lewego lub prawego przycisku myszy

```
if (Input.GetMouseButtonDown(0) || Input.GetMouseButtonDown(1))
{
    Shoot();
}
```

Operator !

Przykład strzału, gdy przycisk nie jest wciśnięty

```
if (!Input.GetMouseButtonDown(0))
{
    Shoot();
}
```

Operator logiczny AND zwróci true tylko i wyłącznie, jeśli oba warunki będą prawdziwe. Należy zauważyć, że kompilator stara się pracować jak najbardziej wydajnie. W związku z tym, jeśli pierwszy sprawdzany warunek nie będzie prawdziwy, to kompilator nie będzie już sprawdzać kolejnego warunku, a od razu zwróci wynik takiej operacji jako false.

Operator logiczny AND && ma wyższy priorytet niż operator OR, dlatego w wyrażeniu złożonym z wielu warunków, taka operacja zostanie obliczona jako pierwsza.

Operator OR || zwraca prawdę, jeśli choć jeden z warunków będzie prawdziwy. W celach optymalizacji, przy pierwszym prawdziwym warunku, kompilator zwróci true, pomijając sprawdzanie pozostałych warunków.

Operator NOT !, zamienia wynik danego wyrażenia z true na false i odwrotnie.

Operacje boolowskie można niejako traktować jak działania matematyczne i chcąc nadać pewnym operacjom priorytet, można posługiwać się nawiasami.

Operator tenarny

Operator warunkowy może zastąpić prostą instrukcję warunkową. Jeżeli spełnione jest wyrażenie logiczne, to operator warunkowy zwraca wyrażenie1, w przeciwnym wypadku zwraca wyrażenie2.

Syntax operatora warunkowego (tenarnego):

```
(wyrażenie logiczne)? wyrażenie1 : wyrażenie2;
```

```
int num = 0;

string name = (num == 0) ? "Alex": "Bob";
```

Switch

Instrukcja switch, podobnie jak instrukcja warunkowa if, steruje programem na podstawie podanej wartości. Zmienna, której wartość będzie testowana może być typu: bool, char, string, int, enum.

Polecenia, które powinny zostać wykonane w ramach jednej wartości case muszą zostać zakończone instrukcją break lub goto case (blok case może kończyć także instrukcja return).

Na końcu można wstawić blok default, który jest opcjonalny.

```
int nr = 0;
switch(nr)
{
      case 0:
      // Run instruction 1
      break;
case 1:
      // Run instruction 2
      break;
case 2:
       // Run instruction 3
      break;
default:
       // Run default instruction
      break;
}
```

Spawnowanie (Instantiate)

W celu stworzenia obiektów na scenie w czasie gry użyjemy metody **Instantiate().**

Za jej pomocą możemy dodać do sceny obiekty (prefaby) w wyznaczonym przez nas miejscu, rotacji i wielu innych parametrach. Metoda Instantiate zawiera wiele przeciążeń, dzięki czemu możemy spawnować obiekty na wiele sposobów.

Przykład poniżej pokazuje jak utworzyć obiekt(prefab) w jego domyślnym położeniu.

```
[SerializeField] GameObject prefab; void Start()
{
    Instantiate(prefab);
}
```

Możemy utworzyć obiekt w wybranym przez nas miejscu i o określonej rotacji za pomocą odpowiedniego przeciążenia metody Instantiate()

```
Instantiate(prefab, new Vector3(4f,-4f,0f), Quaternion.identity);
```

GetComponent

W Unity, dodatkowe funkcjonalności do obiektów dodaje się za pomocą komponentów. To one sprawiają, że dany obiekt w grze staje się kamerą, graczem, przeciwnikiem, itp. Wszystkie komponenty widoczne są w oknie Inspector i każdy z nich ma określone właściwości, które można modyfikować, a także funkcje, które można wyzwalać za pomocą kodu.

W celu zmiany właściwości lub wyzwolenia funkcji dostępnych dla danego komponentu należy odnieść się do niego w skrypcie. Aby to zrobić należy nawiązać do niego referencję za pomocą metody:

```
GetComponent<NazwaKomponentu>()
```

Przykład pobrania pozycji obiektu do którego dołączony jest skrypt:

```
Debug.Log(GetComponent<Transform>().position);
```

W celu zwiększenia wydajności, komponenty należy zapisywać w zmiennych (cachowanie) w metodzie Awake, dzięki czemu powiązanie to robi się tylko 1 raz, np:

```
Rigidbody2D rb = GetComponent<Rigidbody2D>();

SpriteRenderer spriteRend =

GetComponent<SpriteRenderer>();
```

Komponent `Transform` występuje w każdym obiekcie na scenie, dlatego Unity umożliwia pobranie tego komponentu w łatwiejszy sposób, za pomocą aliasu `transform`, który zastępuję instrukcję

```
GetComponent<Transform>()
```

Instrukcję:

`GetComponent<Transform>().position` można zastąpić aliasem:
`transform.position;`

Powyższa możliwość występuje tylko w przypadku komponentu Transform.

Metoda `GetComponent<T>()` zwraca komponent, który jest dołączony do danego obiektu. Istnieją również podobne metody, które zwracają komponenty w złożonych obiektach nadrzędnych i podrzędnych (parent / child), np:

```
Transform trParent = GetComponentInParent<Transform>();

Transform trChildren =

GetComponentInChildren<Transform>();
```

Właściwości

Właściwości można traktować jak rozbudowane zmienne z dodatkowymi funkcjonalnościami. Właściwości mogą być ustawione w taki sposób, by mogły być pobierane z zewnątrz (publiczne), ale ustawiane tylko wewnątrz klasy (prywatne). Jest to możliwe dzięki blokom get i set.

Właściwości nazywa się zgodnie z konwencją PascalCase i mogą być połączone ze zmienną prywatną o takiej samej nazwie (pisanej z małej litery).

```
[SerializeField] private int price;
public int Price
{
      get { return price;}
      private set { price = value;}
}
```

Wewnątrz bloków get i set można zawrzeć dodatkową logikę.

```
[SerializeField] private int price;
public int Price
{
      get { return price; }
      private set {
            if(price >= 0)
            {
                  price = value;
            }
            else
            {
                  price = 0;
            }
      }
}
```

W przypadku, gdy nie ma potrzeby stosowania dodatkowej logiki, można zastosować właściwość automatyczną:

```
public int Price {get; private set;}
```

Unity nie serializuje właściwości, tak jak zmiennych, ale można to zrobić za pomocą atrybutu: [field:SerializeField]

```
[field:SerializeField] public int MyProperty { get; private set; }
```

Tablica

Tablice służą do przechowywania stałej ilości elementów o takim samym typie danych. Elementy w tablicy są indeksowane, oznacza to, że każdy element ma swój numer porządkowy (indeks). Elementy tablicy wywołuje się poprzez numer indeksu podany wewnątrz nawiasu kwadratowego [ ]. Indeksowanie zaczyna się od 0.

W celu przechowania 3 liczb całkowitych możemy stworzyć tablicę o typie int, podając liczbę elementów:

```
int[] liczby = new int[3];
liczby[0] = 12;
liczby[1] = 51;
liczby[2] = 32;
Debug.Log(liczby[0]);   // wyśw. element o indeksie 0
```

Utworzenie tablicy, której wartości będą przypisane w Inspektorze wygląda następująco:

```
[SerializeField] int[] liczby;
```

Przykład tworzenia tablicy inicjalizując ją wartościami, np.:

```
string[] imiona = new string[]{"Kasia", "Basia", "Stasia", "Asia"};
```

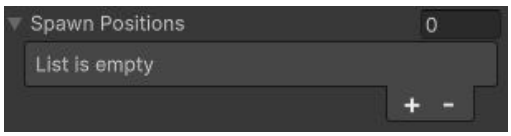
lub krócej:

```
string[] imiona = {"Kasia", "Basia", "Stasia", "Asia"};
```

Przykład poniżej umożliwia ręczne przypisanie obiektów w

Inspektorze, w których ma następować spawnowanie przeciwników.

```
[SerializeField]

Transform[] spawnPositions;
```



Istnieje wiele metod, które zwracają elementy w postaci tablicy. Tablicę można zainicjalizować, poprzez wywołanie takiej metody. Poniżej przykład przypisania do tablicy wszystkich obiektów na Scenie o tagu "Enemy":

```
GameObject[] enemies = GameObject.FindGameObjectsWithTag("Enemy");
```

Przykład przypisania do tablicy wszystkich obiektów ze skryptem <Enemy>

```
Enemy[] enemies = FindObjectsOfType<Enemy>();
```

Operacje na tablicach

W programowaniu bardzo często sprawdza się ilość elementów w tablicy, służy do tego specjalna właściwość Length, np:

```
Debug.Log(liczby.Length); // wyśw. ilość elementów tablicy
```

Tablice możemy sortować dodając bibliotekę System w następujący sposób:

```
Array.Sort(liczby);
```

```
Array.Reverse(liczby);
```

Kopiowanie części tablicy:

```
Array.Copy(a, 2, b, 3, 5);
```

Efektem działania powyższego programu jest skopiowanie pięciu (na co wskazuje ostatni argument metody Copy()) elementów tablicy a do tablicy b. Kopiowanie elementów z tablicy źródłowej rozpocznie się od elementu o indeksie 2, elementy te zostaną wstawione do tablicy docelowej, rozpoczynając od indeksu 3 (pozostałe elementy tablicy b są wypełniane zerami, bo 0 jest wartością domyślną dla elementów tablicy typu int).

Tablica wielowymiarowa

```
int[ , ] tablica2d = new int [3,5]; int[, ,]
tablica3d = new int[2, 2, 2];
```

Inicjalizowanie elementów tablicy dwuwymiarowej:

```
int[,] tab = new int[,]
{
    { 1, 2 },
    { 3, 4 },
    { 5, 6 },
    { 7, 8 }
};
```

lub krócej:

```
int[,] tab = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Zadeklarowana w ten sposób tablica ma rozmiar [4,2] (czyli 4 wiersze i 2 kolumny). Dostęp do poszczególnych elementów takiej tablicy możliwy jest po podaniu wartości obu indeksów, np.:

```
Debug.Log(tab [2,0]); //Wiersz nr 2, kolumna 0 (5)


// GetLength() zwraca dlugosc tablicy 0 - wiersze, 1 - kolumny

Debug.Log(tab.GetLength(0));
```

Wymieszanie tablicy - Algorytm Knuth'a

Czasami zachodzi potrzeba wymieszania elementów tablicy, można to łatwo przeprowadzić z użyciem algorytmu Knuth'a.

```
void Start()
{
    int[] numArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    Shuffle(numArray);
    for (int i = 0; i < numArray.Length; i++)
    {
        Debug.Log(numArray[i]);
    }
}


void Shuffle<T>(T[] a)
{
     for (int i = a.Length - 1; i > 0; i--)
     {
          int rnd = Random.Range(0, i);
          T temp = a[i];
          a[i] = a[rnd];
          a[rnd] = temp;
    }
}
```

Lista

Listy przypominają tablice. Mają tę zaletę, że ilość ich elementów można dowolnie zmieniać w trakcie gry. List użyjemy wtedy, gdy w trakcie gry chcemy dodawać lub usuwać jej elementy.

Listy są dostępne z biblioteki `using System.Collections.Generic;`

Przykład stworzenia listy typu string i dodanie do niej kilku elementów:

```
List<string> auta = new List<string>();

auta.Add("maluch");

auta.Add("trabant");

auta.Add("mustang");

Debug.Log(auta[0]); //wyśw. elementu listy o indeksie 0

auta.Remove("maluch");  //usuwa element "maluch"

auta.RemoveAt(1);       //usuwanie elementu o indeksie 1
```

Pobieranie długości listy (ilości elementów) za pomocą właściwości Count

```
Debug.Log(auta.Count);  // wyświetlanie liczby elementów listy
```

Sortowanie listy

```
auta.Sort(); auta.Reverse();
```

Podobnie, jak w przypadku tablic, listę można inicjalizować za pomocą Inspektora lub za pomocą różnych metod.

Przykład zamiany tablicy na listę i przypisania obiektów na Scenie, które mają na sobie skrypt Enemy:

```
List<Enemy> enemies = FindObjectsOfType<Enemy>().ToList();

Debug.Log(enemies.Count);
```

Pętla foreach

Pętla pozwala wykonywać powtarzalne instrukcje, dopóki podany warunek jest spełniony. Pętla foreach jest używana wraz z kolekcjami iteracyjnymi (tablice, listy, wartości typu string). Pętla foreach iteruje (wykonuje się) tyle razy, ile jest elementów w kolekcji.

Przykład wypisania w konsoli wszystkich elementów tablicy

```
string[] imiona = { "Kasia", "Basia", "Stasia", "Asia" };
foreach (string imie in imiona)
{
    Debug.Log(imie);
}
```

Przykład wypisania w konsoli elementów listy od najmniejszego do największego

```
List<int> liczby = new() { 21, 2, 12, 33 };
liczby.Sort();
foreach (int liczba in liczby)
{
    Debug.Log(liczba);
}
```

Pętla for

Pętla for opiera się na zmiennej iteracyjnej, może operować na kolekcjach, ale nie musi. Pętla for wykonuje się dopóki, podany w nawiasie, warunek jest prawdziwy.

Syntax pętli for:

```
for(zmienna_iteracyjna; warunek; skok)
{
    // instrukcje do wykonania
}
```

zmienna iteracyjna inicjalizuje pętle (w przypadku kolekcji, często inicjalizuje się ją wartością 0, dzięki czemu rozpoczyna się od pierwszego elementu kolekcji)
warunek jest wyznacznikiem tego, kiedy pętla ma zakończyć działanie. Pętla iteruje tak długo, jak warunek pozostaje prawdziwy)
skok modyfikuje wartość zmiennej iteracyjnej. Po każdej iteracji pętli, zmienna iteracyjna zmienia się, aby odnieść się do kolejnych elementów kolekcji i ewentualnie spowodować, że warunek pętli przestanie być prawdziwy, co zakończy działanie pętli

Przykład wypisania liczb od 1 do 10

```
for(int i = 1; i <= 10; i++)
{
    Debug.Log(i);
}
```

Przykład wyświetlania elementów tablicy

```
string[] imiona = { "Kasia", "Basia", "Stasia", "Asia" };
for (int i = 0; i < imiona.Length; i++)
{
    Debug.Log(imiona[i]);
}
```

Pętla while

Pętla while daje podobną swobodę co pętla for. Często używana jest podczas pobierania danych z zewnątrz (z pliku, internetu).

Syntax pętli while:

```
zmienna_iteracyjna;
while(warunek_wykonania_pętli)
{
      //instrukcje;
      skok;
}
```

Przykład wypisania liczb od 5 do 20

```
int i = 5; while(i <= 20)
{
      Debug.Log(i);
      i++;
}
```

Przykład wyświetlania elementów tablicy za pomocą pętli while

```
string[] imiona = { "Kasia", "Basia", "Stasia", "Asia" };
int i = 0; while(i < imiona.Length)
{
    Debug.Log(imiona[i]);     i++;
}
```

Enum

Enum jest typem wyliczeniowym, który pozwala przypisać nazwy do liczb, tak, aby było łatwiej ich używać. Elementy enum można traktować jak stałe, ponieważ definiuje się je podczas tworzenia, a później pozostają niezmienne. W przykładzie poniżej używa się nazw przedmiotów do zebrania zamiast cyfr od 0 do 3 (0 - Ammo, 1 - Weapon, itd...)

```
enum ItemType
{
   Ammo, Weapon, FirstAid, Coin
}

public class EnumExample : MonoBehaviour
{
      void Start()
      {
            ItemType item;
            item = ItemType.Ammo;
            Debug.Log(item);
            Debug.Log((int)item);
      }
}
```

Przykładem użycia enum, może być określenie poziomu trudności gry, np:

```
enum Level { Low, Medium, High }
```

Tuple

Tuple to struktura danych, która może się składać z wielu wartości o różnych typach. Tuple jest przydatny w sytuacji, gdy z funkcji chcemy zwrócić więcej niż jedną wartość.

Przykład tupla:

```
(int, float) nums = (4, 3.5f);
```

Wartości tupli można pobrać za pomocą właściwości Item1, Item2, ...

```
Debug.Log($"{nums.Item1}, {nums.Item2}");
```

Do składowych tupla można dodać specjalne nazwy, dzięki którym można później pobrać daną wartość:

```
(string Desc, float Val) stats = ("attack", 3.5f);
Debug.Log($"{stats.Desc}: {stats.Val}");
```

Struktura (Struct)

Struktura jest typem przypominającym klasę i może być tworzona poza blokiem klasy.

Właściwości Struktury:

- jest typem wartościowym (klasa jest typem referencyjnym)
- nie może dziedziczyć po klasie, ani być przedmiotem dziedziczenia
- może implementować interfejsy
- w strukturze nie można zadeklarować konstruktora bez argumentów,
- składowe struktury nie mogą być inicjalizowane w momencie deklarowania

```
public struct Car
{
	public int power;
	public int mass;
	public string[] colors;
}
```

Deklaracja struct Car z konstruktorem:

```
public struct Car
{
	public int power { get; private set; }
	public int mass { get; private set; }
	public string[] colors;
}

public Car(int power, int mass, string[] colors)
{
	this.power = power;
	this.mass = mass;
	this.colors = colors;
}
```

Tworzenie obiektu o typie Car:

```
Car myCar2 = new Car(550, 1800, new string[] {"White", "Black"} );
```

Słownik (Dictionary)

Słowniki przechowują powiązane pary wartości: klucz i wartość (key, value). Słowniki są bardzo przydatne podczas pracy z formatem JSON, kiedy zachodzi potrzeba zapisu danych na dysku lub pobrania informacji z internetu.

Syntax Dictionary:

```
Dictionary<keyType, valueType> dictName = new
Dictionary<keyType, valueType>();
```

Każdy klucz w słowniku musi być unikalny, to za jego pomocą będzie wywoływana połączona z nim wartość. Klucze mogą być innego typu niż sama wartość. Klucze nie mogą być zmieniane.

Przykład słownika ze statystykami gracza:

```
Dictionary<string, int> playerStats = new Dictionary<string, int>()
{
    {"attack", 12},
    {"mana", 50},
    {"speed", 6}
};
```

Wartości wywołuje się poprzez nazwę klucza:

```
playerStats["attack"]
```

W każdym momencie można dodać lub usunąć dany element ze słownika.

```
playerStats.Add("enemyKilled", 23);
playerStats.Remove("enemyKilled");
```

Zmiana wartości danego klucza:

```
playerStats["enemyKilled"] = 1;
```

Właściwość Count zwraca ilość elementów w słowniku:

```
playerStats.Count
```

Wywołanie wartości ze słownika:

```
foreach (var stat in playerStats.Values)
{
    Debug.Log(stat);
}
```

Wywołanie kluczy i wartości ze słownika:

```
foreach (KeyValuePair<string, int> stat in playerStats)
{
    Debug.Log(stat.Key + ": " + stat.Value);
}
```

Rzutowanie (casting)

W programowaniu zdarza się potrzeba zamiany jednego typu na inny.

Rzutowanie niejawne (automatyczne)

Mniejsze typy można rzutować automatycznie na większe, dzięki rzutowaniu niejawnemu (Implicit Casting) `byte -> int -> long -> float -> double`

Rzutowanie niejawne (automatyczne)

```
byte greenChannel = 12;

int num = greenChannel;
```

Rzutowanie jawne (manualne)

Większe typy można rzutować na mniejsze, ale ponieważ może wystąpić zapisanie większej wartości niż to możliwe, należy manualnie wykonać rzutowanie jawne(Explicit Casting) przez podanie typu w nawiasie: (char), (byte), (int)

```
int randomNumber = 12;

byte greenChannel = (byte)randomNumber;
```

Parsowanie stringów

Programy lub gry często przyjmują informacje od użytkownika.

Cokolwiek zostanie wpisane, zawsze jest traktowane jako tekst, nawet, jeśli jest to liczba.

Chcąc traktować wprowadze dane jako liczbę należy ją sparsować, np:

```
string input = "12";

int num = int.Parse(input);
```

Powyższa metoda wywoła błąd, jeżeli podany tekst nie będzie mógł być przetłumaczony na liczbę (np. "aaa" , "01b") W tym celu należy użyć metody `TryParse()`

```
string input = "12";
int num;
if(!int.TryParse(input, out num))
{
    Debug.Log("Can't parse");
}
else
{
    Debug.Log(num);
}
```

String Builder

StringBuilder używany jest wtedy, gdy potrzebujemy wielokrotnie nadpisywać zmienną typu string.

StringBuilder korzysta z biblioteki `using System.Text;`

```
StringBuilder sb = new StringBuilder();
sb.Append("Hello"); sb.Append("!");
sb.Insert(5, " ");
Debug.Log(sb);
```

Wykorzystanie StringBuildera w pętli:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 100; i++)
{
      sb.Append(i + ", ");
}
Debug.Log(sb);
```

Przypisanie StringBuildera do zmiennej typu string:

```
string myString = sb.ToString();
```

Więcej o StringBuilder: https://learn.microsoft.com/en-us/dotnet/api/system.text.stringbuilder?view=net-7.0

Typy wartościowe i referencyjne

Typami wartościowymi w C# i Unity są: int, float, byte, char, bool, enum, struct, Vector2/3, Color32. Podczas przypisywania lub przekazywania zmiennej, tworzona jest jej kopia, której zmiana nie powoduje zmian oryginału.

```
void Start()
{
      int healthPoints = 12;
      int hp = healthPoints;
      healthPoints = 20;
      Debug.Log(healthPoints);
      Debug.Log(hp);
}
```

lub

```
void Start()
{
      int healthPoints = 12;
      IncreaseHp(healthPoints);
}

void IncreaseHp(int hp)
{
      hp += 1;
      Debug.Log(hp);
}
```

Typami referencyjnymi w C# i Unity są: obiekty, tablice, listy, komponenty.

Zmienna typu referencyjnego zawiera w sobie jedynie adres do miejsca w pamięci, w którym znajduje się konkretna wartość.

Podczas przypisywania lub przekazywania zmiennej, nowa zmienna dalej wskazuje to samo miejsce w pamięci.

```
void Start()
{
    SpriteRenderer spriteRenderer =
GameObject.FindGameObjectWithTag("Player").GetComponent<SpriteRenderer>();
    ChangeColor(spriteRenderer);
}

void ChangeColor(SpriteRenderer sp)
{
     sp.color = Color.red;
}
```

Pamięć na stosie zarządzana jest automatycznie przez system, natomiast pamięć na stercie zarządzana jest przez tzw. Garbage Collector (GC). Jest to specjalne narzędzie do usuwania obiektów, do których nie prowadzą żadne powiązania ze stosu.

Ref

Typy wartościowe można przekazywać do funkcji traktując je jak typy referencyjne za pomocą słowa kluczowego ref.
Zmienna a, przekazywana jest do funkcji przez referencję, dzięki czemu po jej wykonaniu jej wartość ulega zmianie.

```
void Start()
{
      var a = 12;
      IncreaseNumber(ref a);
      Debug.Log($"{a}");
}


void IncreaseNumber(ref int _a)
{
    _a += 1;
}
```

Generic <T>

Typy generyczne zastępują wszystkie możliwe typy, dzięki czemu ich użycie staje się bardzo wszechstronne.

Poniżej przykład generycznej funkcji, która zamienia wartości dwóch zmiennych jakiegokolwiek typu.

```
void Start()
{
	var a = "Anna";
	var b = "Mark";
	Swap(ref a, ref b);
	Debug.Log($"{a}, {b}");
}

void Swap<T>(ref T _a, ref T _b)
{
	T temp;
	temp = _a;
	_a = _b;
	_b = temp;
}
```

Dziedziczenie klas

Klasy służą do tworzenia obiektów. W przypadku, gdy obiekty mają wspólne cechy, można zastosować klasę bazową opisującą cechy wspólne, a następnie stworzyć klasy pochodne dla każdego obiektu osobno.
Daje to możliwość łatwiejszego zarządzania kodem i kategoryzuje obiekty o cechach wspólnych.

Przykładem może być stworzenie klasy bazowej Gun dla broni strzelających jak: Pistol, MachineGun, Rifle, itd.

Przykład klasy bazowej Gun.

```
using UnityEngine;
public class Gun: MonoBehaviour
{
      [field:SerializeField] public int Attack {get; set;}
      [field: SerializeField] public int MagCapacity {get; set;}
      public void UseGun()
      {
            Debug.Log("Shoot");
      }
}
```

Klasa pochodna

Klasy pochodne automatycznie mają dostęp do elementów klasy bazowej. Dodatkowo mogą rozszerzać klasę bazową o własne pola, metody, itp.

```
public class MachineGun : Gun
{
	[field: SerializeField] public int AmmoInSeries {get; set;}
	void Start()
	{
		UseGun();
	}
}
```

Klasy pochodne mogą nadpisywać metody, tak, aby dostosować ich działanie do swoich potrzeb.

```
public class Pistol : Gun
{
	void Start()
	{
		UseGun();
	}
	new void UseGun()
	{
	  Debug.Log($"shoot from pistol with {Attack} and magcap
{MagCapacity}");
	}
}
```

Klasa abstrakcyjna

Klasa abstrakcyjna to taka, która nie może tworzyć obiektów.

Zazwyczaj stanowi ona bazę dla klas pochodnych.

Wewnątrz klasy można tworzyć również abstrakcyjne funkcje i właściwości. W takim przypadku nie podaje się ciała funkcji, ponieważ musi być ona nadpisana w klasie pochodnej.

```
using UnityEngine;
public abstract class Gun : MonoBehaviour
{
    [field: SerializeField] public int Attack { get; set; }
    [field: SerializeField] public int MagCapacity { get; set; }
    [field: SerializeField] public int Price { get; set; }
    public abstract void UseGun(); //musi być nadpisana w klasie pochodnej
(override)
    public void Reload()
    {
        // może, ale nie musi być nadpisana w klasie pochodnej
     }
}
```

Klasa pochodna od klasy abstrakcyjnej musi zawierać implementację metod abstrakcyjnych. using UnityEngine;

```
public abstract class Gun : MonoBehaviour
{
      public abstract void UseGun();
}

public class Pistol : Gun
{
      public override void UseGun()
      {
            Debug.Log($"Shoot with pistol {Attack}, magcap:
{MagCapacity}");
      }
}
```

Interfejsy

Interfejs jest klasą w pełni abstrakcyjną (nie można z niego stworzyć obiektu), dlatego też nie może posiadać konstruktora.
Implementacja interfejsu daje pewność, że w danej klasie będą dostępne wszystkie właściwości i metody, określone w interfejsie.
Wewnątrz interfejsu mogą znaleźć się tylko właściwości i puste metody.
Przed nazwą interfejsu dodaje się wielką literę I.
Elementy interfejsu są domyślnie abstrakcyjne i publiczne.
Interfejsy implementuje się za pomocą dwukropka :
Można implementować wiele interfejsów, które oddziela się przecinkami.
W przypadku interfejsów (w odróżnieniu od zwykłej klasy abstract) nie trzeba używać słowa kluczowego override.
Klasy implementujące interfejs muszą zawierać jego wszystkie metody i właściwości.
Wiele klas może implementować ten sam interfejs. Dzięki takiemu podejściu wywoływanie funkcji lub właściwości można przeprowadzać poprzez nazwę interfejsu, a niekoniecznie przez nazwę klasy.

Tworzenie interfejsu do zbierania itemków w grze.
Klasa implementująca interfejs będzie na pewno posiadała odpowiednie właściwości i metodę do pobrania itemka.

```
public enum ItemType {gun, ammo, hp, granade }

public interface ICollectable
{
      ItemType ItemCollectType { get; set; }
      int Quantity { get; set; }
      (ItemType, int) Collect();
}
```

Implementacja interfejsu:

```
using UnityEngine;
public class CollectableItem : MonoBehaviour, ICollectable
{
      [field: SerializeField] public ItemType ItemCollectType { get; set; }
      [field: SerializeField] public int Quantity { get; set; }
      public (ItemType, int) Collect()
      {
            return (ItemCollectType, Quantity);
      }
}
```

Obiekty statyczne

Elementy statyczne odnoszą się do danej klasy, a nie obiektu.
Przykład poniżej dodany do skryptu <Enemy> zliczy ilość wszystkim przeciwników utworzonych w grze.

```
public class Enemy : Npc
{
	public static int enemyCount = 0;

	void Start()
	{
		enemyCount++;
	}

	public static int GetEnemyCount()
	{
		return enemyCount;
	}
}
```

Funkcje statyczne odwołują się do całej klasy, dzięki czemu można je wywołać bez nawiązywania referencji do obiektu.

Wywołanie funkcji:

```
int enemyCount = Enemy.GetEnemyCount();
```

Wzorzec Singleton

Singletony pozwalają na bezpośredni dostęp do danego skryptu bez tworzenia referencji. Singleton oznacza, że na scenie będzie tylko jeden obiekt z takim skryptem.

```
public class UIManager : MonoBehaviour
{
	public static UIManager Instance {get; private set;}
	private void Awake()
	{
		if (Instance != null && Instance != this)
		{
			Destroy(this);
		}
		Else
		{
			Instance = this;
		}
	}

	public void UpdateUI(int hp)
	{
		// display hp
	}
}
```

Wywołanie funkcji ze skryptu Singletona:

```
UIManager.Instance.UpdateUI(12);
```

Zasady SOLID

SOLID to zestaw zasad projektowych stosowany przy programowaniu obiektowym.

Służą do tworzenia bardziej elastycznego, modułowego i łatwego do utrzymania kodu. Ich stosowanie pomaga unikać nadmiernego powiązania między klasami, ułatwia testowanie i rozwijanie kodu oraz zapobiega “efektowi domina”, w którym zmiana w jednym miejscu powoduje konieczność modyfikacji wielu innych miejsc w kodzie.

1. Single responsibility principle,
2. Open-closed principle,
3. Liskov substitution principle,
4. Interface segregation principle,
5. Dependency inversion principle

Single Responsibility Principle

Zasada pojedynczej odpowiedzialności: Każda klasa powinna mieć tylko jedną odpowiedzialność. Oznacza to, że klasa powinna być odpowiedzialna za realizację jednego, dobrze zdefiniowanego zadania, co ułatwia jej ponowne użycie i utrzymanie.

Open-Closed Principle

Zasada otwarte-zamknięte. Kod powinien być otwarty na rozszerzenie, ale zamknięty na modyfikację. Oznacza to, że istniejące zachowanie klasy powinno być łatwe do rozszerzenia przez dodanie nowych funkcjonalności, bez konieczności modyfikowania istniejącego kodu.

Liskov Substitution Principle

Zasada podstawiania Liskov. Obiekty danego typu powinny być zastępowalne przez obiekty ich podtypów, nie wpływając na poprawność programu. W praktyce

oznacza to, że należy dbać o to, aby każdy podtyp dziedziczący po pewnym typie zachowywał się zgodnie z oczekiwaniami tego typu.

Interface Segregation Principle

Zasada segregacji interfejsów. Klienci nie powinni być zmuszani do zależności od interfejsów, których nie używają. Oznacza to, że interfejsy powinny być odpowiednio podzielone na mniejsze, bardziej sprecyzowane interfejsy, dostosowane do konkretnych potrzeb klientów.

Dependency Inversion Principle

Zasada odwrócenia zależności. Moduły powinny polegać na abstrakcjach, a nie na konkretnych implementacjach. Oznacza to, że zależności między modułami powinny być oparte na abstrakcyjnych interfejsach lub klasach bazowych, a nie na konkretnych klasach.

Kolejność operatorów

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Relational	< <= => >	Left to right
Equality	== !=	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=	Right to left
Comma	,	Left to right