OKIEM EKSPERTA

Stwórz grę w Unity, a nauczysz się programowania w C#!

Pisanie kodu, które sprawia radość

Wydanie VIII

Harrison Ferrone

Helion

<packt>

Tytuł oryginału: Learning C# by Developing Games with Unity 6: Get to grips with coding in C# and build simple 3D games in Unity from the ground up, 8th Edition

Tłumaczenie: Radosław Meryk, Piotr Pilch

ISBN: 978-83-289-4178-6

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/swtgr8
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Bloki budulcowe programowania

Rozdział 2

Każdy język programowania, gdy zetkniesz się z nim po raz pierwszy, wygląda jak starożytna greka. Język C# nie jest wyjątkiem. Dobra wiadomość jest jednak taka, że poza tą początkową tajemniczością wszystkie języki programowania składają się z tych samych podstawowych bloków budulcowych. Zmienne, metody i klasy (lub obiekty) to DNA konwencjonalnego programowania. Zrozumienie tych prostych pojęć otwiera cały świat zróżnicowanych i złożonych zastosowań. Tak samo jest z ludźmi — chociaż we wszystkich mieszkańcach Ziemi są tylko cztery różne nukleobazy DNA, to każdy z nas jest unikatowym organizmem.

Jeśli jesteś nowicjuszem w programowaniu, w tym rozdziale znajdziesz wiele przydatnych informacji, których poznanie może wywrzeć wpływ na Twoje pierwsze w życiu wiersze kodu. Nie chodzi mi jednak o to, aby przeciążać Twój mózg faktami i liczbami. Ten rozdział ma Ci dać holistyczny wgląd w bloki budulcowe programowania z wykorzystaniem przykładów z codziennego życia.

Ten rozdział prezentuje ogólny widok elementów, z których składa się program. Zapoznanie się ze sposobem współdziałania poszczególnych elementów przed bezpośrednim przystąpieniem do kodowania nie tylko pomoże początkującym programistom postawić pierwsze kroki, ale również, dzięki łatwym do zapamiętania porównaniom, pozwoli utrwalić zdobyte wiadomości. Dość marudzenia. W tym rozdziale skoncentruję się na następujących tematach:

- Definiowanie zmiennych.
- Przeznaczenie metod.
- Wprowadzenie w tematykę klas.
- Korzystanie z komentarzy.
- Wykorzystanie bloków budulcowych programowania w praktyce.

Zacznijmy!

Definiowanie zmiennych

Na początek trzeba zadać sobie proste pytanie: czym jest zmienna? W zależności od punktu widzenia można na nie odpowiedzieć na kilka różnych sposobów (wszystkie są poprawne):

- **Pojęciowo** — zmienna jest najbardziej podstawową jednostką programowania — tym samym, czym atom jest dla świata fizycznego. Wszystko zaczyna się od zmiennych, a programy nie mogą bez nich istnieć.
- **Formalnie** — zmienna jest niewielkim fragmentem pamięci komputera, w której jest przechowywana przypisana wartość. Każda zmienna określa miejsce, gdzie jest zapisana związana z nią informacja (ta lokalizacja nazywa się adresem pamięci), a także jaka jest jej wartość i typ danych (na przykład liczby, słowa lub listy).
- **Praktycznie** — zmienna jest kontenerem. Możesz stworzyć nową zmienną, kiedy chcesz, wypełnić ją informacjami, przenieść w inne miejsce, zastąpić to, co zawiera, i odwoływać się do niej w razie potrzeby. Zmienne mogą być przydatne nawet wtedy, gdy są puste.

Szczegółowe objaśnienie zmiennych można znaleźć w dokumentacji Microsoftu dotyczącej języka C#, pod adresem *https://learn.microsoft.com/pl-pl/dotnet/csharp/language-reference/language-specification/variables*.

Analogią do zmiennej w życiu jest skrzynka pocztowa. Pamiętasz ją (rysunek 2.1)?

Rysunek 2.1. Rząd kolorowych skrzynek pocztowych

Uwaga

Aby zaprezentować pełny widok programu Unity Editor, wszystkie zrzuty ekranu zostały wykonane w trybie pełnoekranowym. Kolorowe wersje wszystkich rysunków zamieszczonych w książce są dostępne pod adresem *https://ftp.helion.pl/przyklady/swtgr8.zip*.

Mogą znaleźć się w niej listy, rachunki, zdjęcie ciotki Marysi — cokolwiek. Chodzi o to, że w skrzynce pocztowej mogą być przechowywane różne rzeczy — mogą mieć nazwy, zawierać informacje (poczta fizyczna), a ich zawartość może się zmieniać (jeśli masz odpowiednie uprawnienia). Podobnie zmienne mogą zawierać różne rodzaje informacji.

Zmienne w języku C# mogą zawierać ciągi znaków (tekst), liczby całkowite (liczby), a nawet wartości logiczne (wartości binarne reprezentujące prawdę lub fałsz).

Nazwy są ważne

Wróćmy do zdjęcia przedstawionego na rysunku 2.1. Gdybym Cię poprosił, abyś podszedł do skrzynki pocztowej i ją otworzył, pierwszą rzeczą, o którą prawdopodobnie byś zapytał, byłoby: Którą? Gdybym odpowiedział, że rodzinną skrzynkę Kowalskich albo czerwoną skrzynkę pocztową, albo okrągłą skrzynkę pocztową, to zyskałbyś niezbędny kontekst do tego, aby otworzyć tę skrzynkę pocztową, o którą mi chodzi. Na podobnej zasadzie, gdy tworzysz zmienne, powinieneś nadawać im niepowtarzalne nazwy, do których można się później odwołać. Specyfiką odpowiedniego formatowania i nadawania opisowych nazw zajmiemy się w rozdziale 3., a na razie nie komplikujmy niczego.

Zmienne pełnią funkcję symboli zastępczych

Gdy tworzysz zmienną i nadajesz jej nazwę, w gruncie rzeczy tworzysz symbol zastępczy (*placeholder*) wartości, którą chcesz przechowywać. Dla przykładu rozważmy następujące proste równanie arytmetyczne:

```
2+9=11
```

Nie ma tu niczego tajemniczego. Co jednak zrobić, aby liczba `9` była zmienną? Rozważmy następujący blok kodu:

```
mojaZmienna = 9
```

Teraz możemy użyć nazwy zmiennej `mojaZmienna` jako substytutu wartości `9` wszędzie, gdzie tego potrzebujemy:

```
2 + mojaZmienna = 11
```

Jeśli zastanawiasz się, czy wykorzystaniem zmiennych rządzą specjalne zasady, odpowiedź brzmi: „Tak”. Omówimy je dokładniej w rozdziale 3., więc na razie się nimi nie martw.

Chociaż ten przykład nie jest realnym kodem w C#, ilustruje moc zmiennych i ich zastosowanie w roli symboli zastępczych.

Dość teorii. Zdefiniujmy w skrypcie *NaukaProgramowania*, który stworzyliśmy w rozdziale 1., prawdziwą zmienną:

1. Kliknij dwukrotnie skrypt *NaukaProgramowania.cs* w oknie *Project* w Unity, aby otworzyć go w Visual Studio Code.
2. Wprowadź odstęp między *wierszami* 4. i 5., po czym, aby zadeklarować nową zmienną, dodaj następujący wiersz kodu:

   ```
   public int CurrentAge = 30;
   ```

3. Wewnątrz nawiasów metody Start dodaj dwa logi diagnostyczne, aby wydrukować wyniki obliczeń:

```
Debug.Log(30 + 1);
Debug.Log(CurrentAge + 1);
```

Przeanalizujmy kod, który właśnie dodałeś. Najpierw utworzyliśmy nową zmienną, o nazwie `CurrentAge`, i przypisaliśmy jej wartość `30`. Następnie dodaliśmy dwa logi diagnostyczne, aby wydrukować wyniki obliczeń `30 + 1` i `CurrentAge + 1`. W ten sposób pokazałem, w jaki sposób zmienne przechowują wartości. Zmiennych można używać dokładnie w taki sam sposób, w jaki używamy wartości.

Należy również zwrócić uwagę, że zmienne publiczne wyświetlają się w oknie *Unity Inspector*, podczas gdy prywatne się w nim nie wyświetlają. Na razie nie martw się o składnię — po prostu zadbaj o to, aby Twój skrypt wyglądał tak samo jak skrypt pokazany na rysunku 2.2.

LearningCurve.cs ×

Assets > Scripts > LearningCurve.cs > LearningCurve

```
1   using UnityEngine;
2
    0 references
3   public class LearningCurve : MonoBehaviour
4   {
        1 reference
5       public int CurrentAge = 30;
6
7       // Start is called once before the first execution of Update after the MonoBehaviour is created
        0 references
8       void Start()
9       {
10          Debug.Log(30 + 1);
11          Debug.Log(CurrentAge + 1);
12      }
13
14      // Update is called once per frame
        0 references
15      void Update()
16      {
17
18      }
19  }
```

Rysunek 2.2. Skrypt C# NaukaProgramowania.cs otwarty w Visual Studio Code

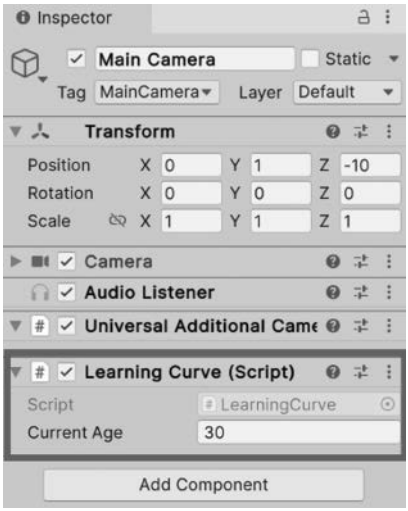
Aby zakończyć edycję kodu, zapisz plik za pomocą opcji *Plik/Zapisz* lub dowolnej kombinacji klawiszy obsługiwanej przez Twój komputer. Zapisywanie jest kluczowym krokiem podczas edytowania skryptów, ponieważ Unity rozpoznaje tylko te zmiany, które zostały zapisane w edytorze. Jeśli dodasz kod do skryptu w Visual Studio Code, ale go nie zapiszesz, Twoje zmiany nie będą widoczne w Unity.

Aby skrypty mogły działać w Unity, muszą być dodane do kolekcji `GameObjects` na scenie. Unity traktuje wszystko w Twojej grze — światła, awatary graczy, przedmioty, budynki i inne — jako obiekty `GameObject`.

Domyślnie przykładowa scena w grze *Narodziny bohatera* ma kamerę do renderowania sceny i światło kierunkowe do jej oświetlania. Zatem dla uproszczenia dodajmy skrypt *NaukaProgramowania* do kamery:

Przeciągnij i upuść skrypt *NaukaProgramowania.cs* na obiekt *Main Camera*.

1. Wybierz pozycję *Main Camera* tak, by wyświetliła się w panelu *Inspector*, i zadbaj o to, by komponent *NaukaProgramowania.cs* (skrypt) został prawidłowo podłączony (rysunek 2.3).



Rysunek 2.3. Skrypt NaukaProgramowania podłączony do panelu Inspector

2. Kliknij *Play* widoczny w górnej środkowej części edytora Unity i obserwuj wynik w panelu *Console* (rysunek 2.4).

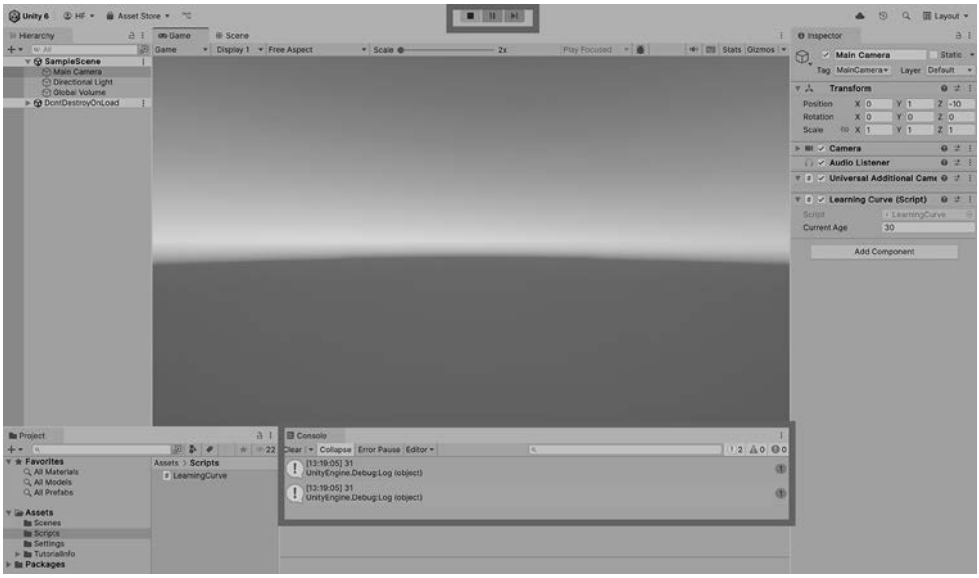
Być może zauważyłeś, że edytor ma nieco ciemniejszy odcień, a przycisk *Play* po uruchomieniu gry zmienił kolor na niebieski. Dzieje się tak dlatego, że środowisko Unity ma dwa stany: edytor i środowisko wykonawcze. Kiedy pracujesz nad skryptami lub dodajesz obiekty do swojej sceny, pracujesz w stanie edytora.

Wszystkie zmiany wprowadzone w tym stanie zostaną zapisane w projekcie. Kiedy jednak naciśniesz przycisk *Play*, Unity przełączy się do stanu wykonawczego. Wszelkie zmiany wprowadzone podczas działania gry nie zostaną zapisane w projekcie, więc zwróć szczególną uwagę na aktualizacje.

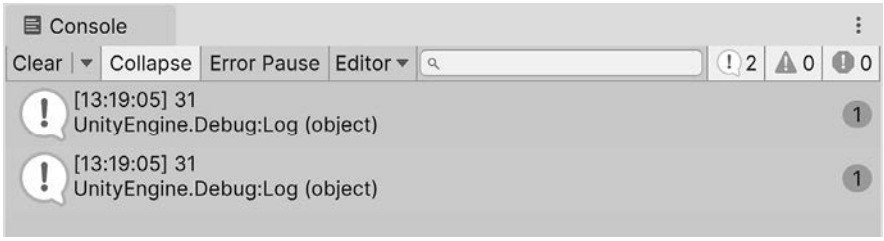
Instrukcje `Debug.Log()` spowodowały wyświetlenie wyniku prostych równań matematycznych umieszczonych pomiędzy nawiasami. Jak widać na zrzucie ekranu z panelem *Console* (rysunek 2.5), równanie, które wykorzystywało zmienną, działało tak samo, jak gdyby zastosowano w nim liczbę.

Opis sposobu, w jaki Unity przekształca skrypty C# na komponenty, zamieszczę w punkcie „Skrypty stają się komponentami” na końcu tego rozdziału. Najpierw jednak pokażę, jak można zmienić wartość jednej ze zdefiniowanych zmiennych.

Ponieważ jak widać na rysunku 2.2, w *wierszu* 5. zadeklarowałeś `CurrentAge` jako zmienną publiczną, przechowywaną przez nią wartość można zmienić w skrypcie lub



Rysunek 2.4. Okno edytora Unity z objaśnieniami do przeciągania i upuszczania skryptów



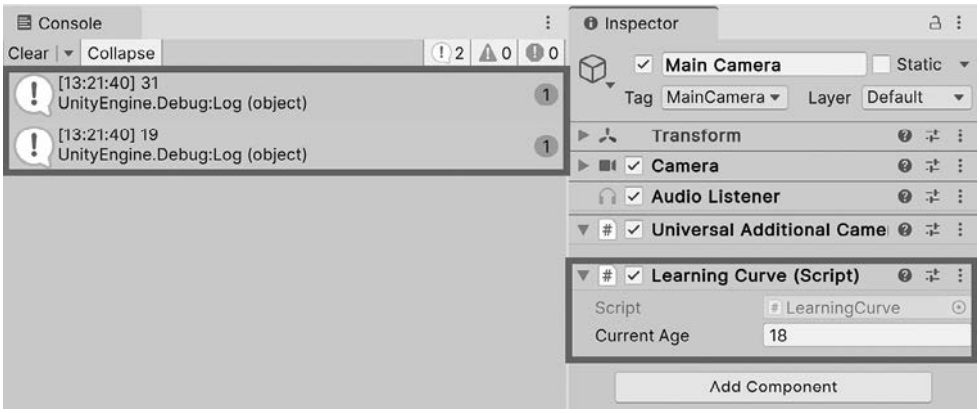
Rysunek 2.5. Konsola Unity z wyjściem diagnostycznym z dołączonego skryptu

w oknie *Unity Inspector*. Następnie we wszystkich miejscach w kodzie, gdzie została użyta zmienna, będzie wykorzystana jej zaktualizowana wartość. Zobaczmy, jak to działa w praktyce:

1. Jeśli scena nadal działa, zatrzymaj grę kliknięciem przycisku *Play*.
2. W panelu *Inspector* zmień wartość *Current Age* na 18.
3. Ponownie odtwórz scenę i obserwuj nowe dane wyjściowe w panelu *Console* (rysunek 2.6).

Pierwszy wynik będzie wynosić 31, ponieważ niczego w skrypcie nie zmieniliśmy, ale drugi będzie wynosił 19, zmieniliśmy bowiem wartość zmiennej `CurrentAge` w oknie *Inspector*.

Celem pokazanego przykładu nie było szczegółowe omówienie składni zmiennych, ale zaprezentowanie, że zmienne działają jak kontenery, które można utworzyć raz, by następnie odwoływać się do nich w innym miejscu.



Rysunek 2.6. Konsola Unity z komunikatami diagnostycznymi i skryptem NaukaProgramowania dołączonym do głównej kamery

Teraz, gdy wiesz, jak tworzyć zmienne w języku C# i przypisywać do nich wartości, nadszedł czas, by przejść do kolejnego ważnego bloku budulcowego programowania: metod!

Metody

Zmienna sama w sobie służy wyłącznie do śledzenia przypisanej do niej wartości. Chociaż ma to kluczowe znaczenie, same zmienne nie wystarczą do tworzenia złożonych aplikacji. Co zatem zrobić, aby tworzyć operacje i implementować zachowania w kodzie? Krótka odpowiedź brzmi: należy skorzystać z metod.

Ważna uwaga

Zanim dotrzemy do tego, czym są metody i jak z nich korzystać, warto się zapoznać z niezbędną terminologią. W świecie programowania powszechnie występują pojęcia *metoda* i *funkcja*, które są używane zamiennie, zwłaszcza w kontekście Unity.

Ponieważ C# jest językiem obiektowym (to zagadnienie opiszę w rozdziale 5.), w pozostałej części tej książki będę się posługiwać terminem *metoda*, co jest zgodne ze standardowymi wytycznymi obowiązującymi dla języka C#.

Gdy w dokumentacji *Scripting Reference* lub dowolnej innej dokumentacji napotkasz termin *funkcja*, możesz przyjąć, że chodzi o *metodę*.

Metody definiują działania

Podobnie jak w przypadku zmiennych definicja metod w programowaniu może być bardzo rozwlekła lub niebezpiecznie krótka. Oto trzy podejścia do rozważenia:

- *Pojęciowo* — metody opisują sposoby wykonywania działań w aplikacjach.
- *Formalnie* — metoda jest blokiem kodu zawierającym wykonywalne instrukcje, które są uruchamiane, gdy metoda zostaje wywołana za pomocą jej nazwy.

Metody mogą przyjmować argumenty (zwane również parametrami), które mogą być wykorzystane wewnątrz zasięgu metody.

- *Praktycznie* — metoda jest kontenerem dla zestawu instrukcji, które są uruchamiane za każdym razem, gdy metoda zostaje wywołana. Te kontenery także mogą przyjmować zmienne jako dane wejściowe. Do zmiennych przekazanych jako parametry można się odwoływać tylko wewnątrz metody.

Ogólnie rzecz biorąc, metody stanowią szkielet każdego programu — łączą kod ze sobą. Prawie wszystko w programach jest zbudowane z metod.

Szczegółowy przewodnik po metodach można znaleźć w dokumentacji Microsoftu dotyczącej C# pod adresem *https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods.*

Metody to także symbole zastępcze

W celu wyjaśnienia koncepcji metod rozważmy uproszczony przykład dodawania dwóch liczb. Gdy piszesz skrypt, to w gruncie rzeczy układasz wiersze kodu, które komputer powinien uruchomić po kolei. Gdy po raz pierwszy musisz dodać do siebie dwie liczby, możesz po prostu zrobić to tak jak w poniższym bloku kodu:

```
pierwszaLiczba + innaLiczba
```

Później jednak zauważysz, że w innym miejscu również musisz dodać do siebie dwie liczby.

Zamiast kopiowania i wklejania tego samego wiersza kodu, co skutkuje powstaniem kodu „spaghetti" i czego należy za wszelką cenę unikać (za każdym razem gdy powtarzasz kod, zwielokrotniasz miejsce, w którym będzie trzeba dokonywać aktualizacji po pojawieniu się zmian w przyszłości), możesz utworzyć metodę, która zajmie się tym działaniem:

```
DodajLiczby()
{
    pierwszaLiczba + innaLiczba;
}
```

Teraz metoda `DodajLiczby`, podobnie jak zmienna, jest symbolem zastępczym w pamięci. Jednak zamiast wartości zawiera blok instrukcji. Gdy w dowolnym miejscu skryptu posłużysz się nazwą metody (czyli ją wywołasz), spowoduje to natychmiastowe wstawienie zapisanych w niej instrukcji. Nie musisz w tym celu powtarzać żadnego kodu.

```
DodajLiczby()
```

Jeśli odkrywasz, że w kółko wpisujesz te same wiersze kodu, to prawdopodobnie nie zauważyłeś, iż można uprościć powtarzane działania i umieścić je w metodach.

Ciągłe wpisywanie tego samego kodu programiści żartobliwie nazywają tworzeniem **kodu spaghetti**. Być może słyszałeś również o stosowanej przez nich zasadzie **DRY** (*Don't Repeat Yourself* — dosłownie: nie powtarzaj się), którą powinieneś powtarzać jak mantrę.

Po zaprezentowaniu nowego pojęcia w pseudokodzie najlepiej zaimplementować je samodzielnie. Aby oswoić pojęcie metod, zrobimy to w następnym podrozdziale, „Klasy — wprowadzenie".

Otwórzmy ponownie skrypt *NaukaProgramowania* i zobaczmy, jak działają metody w języku C#. Podobnie jak w przypadku przykładu użycia zmiennych skopiuj kod do skryptu, aby wyglądał dokładnie tak jak na rysunku 2.7. Dla zwięzłości usunąłem ze skryptu poprzedni przykład kodu. Oczywiście możesz go pozostawić:

1. Otwórz w Visual Studio Code skrypt *NaukaProgramowania*.
2. Dodaj w *wierszu* 6. nową zmienną:

```
public int YearsToAdd = 1;
```

3. W *wierszu* 15. dodaj nową metodę, która dodaje do siebie zmienne `CurrentAge` i `YearsToAdd` oraz wyświetla wynik:

```
void ComputeAge()
{
    Debug.Log(CurrentAge + YearsToAdd);
  }
```

4. Kod wewnątrz metody `Start` zastąp następującym wierszem kodu, który wywołuje nową metodę `ComputeAge`:

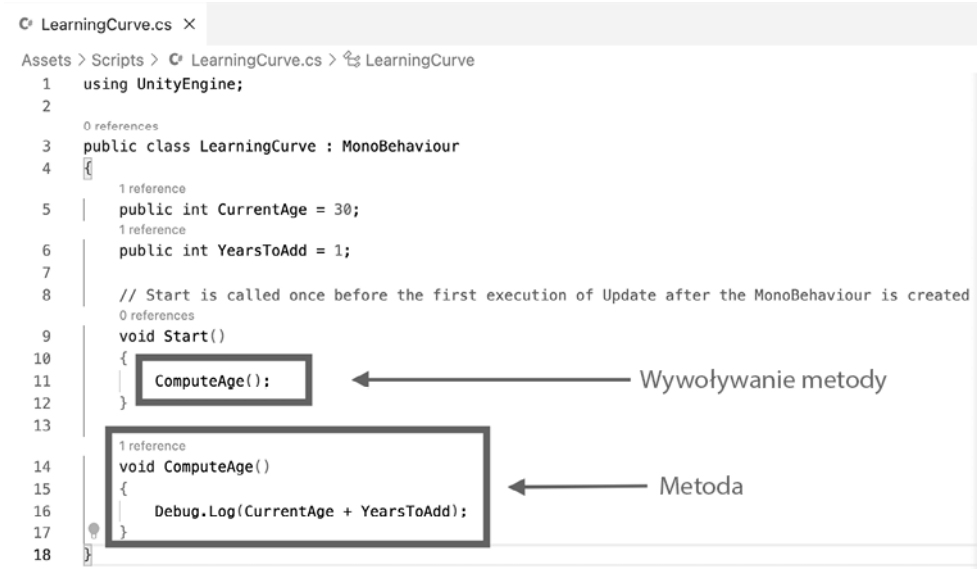
```
void Start()
{
    ComputeAge();
}
```

5. Zanim uruchomisz skrypt w Unity, sprawdź, czy Twój kod wygląda tak jak na rysunku 2.7.
6. Zapisz plik, a następnie w Unity naciśnij *Play*, aby w panelu *Console* zobaczyć nowy wynik.

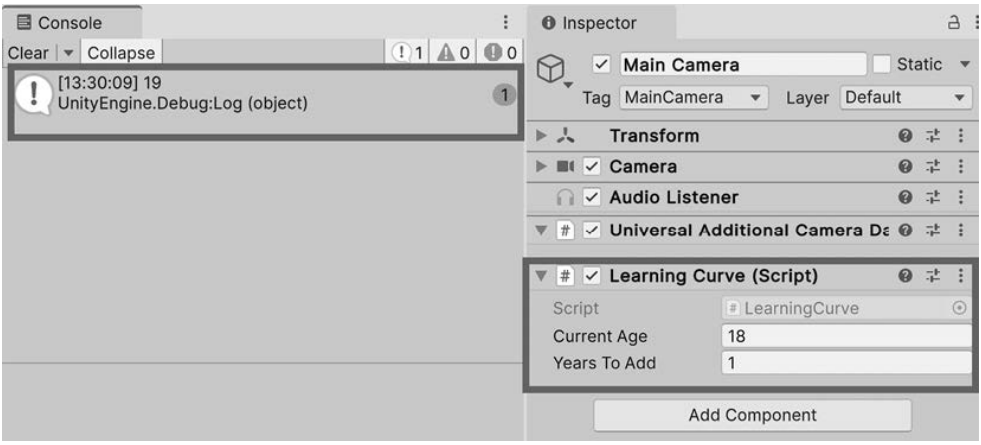
W *wierszach* od 14. do 17. zdefiniowałeś swoją pierwszą metodę i wywołałeś ją w *wierszu* 11. Teraz wszędzie, gdziekolwiek zostanie wywołana metoda `ComputeAge()`, zostaną do siebie dodane dwie zmienne, a wynik zostanie wyświetlony na konsoli (nawet gdy wartości zmiennych się zmienią). Zapamiętaj, że w oknie *Unity Inspector* ustawiłeś zmienną `CurrentAge` na 18, a wartość w oknie *Inspector* zawsze zastępuje wartość ustawioną w skrypcie C# (rysunek 2.8).

Za pomocą panelu *Inspector* wypróbuj różne wartości zmiennych, aby zobaczyć działanie metody w praktyce! Więcej informacji na temat składni kodu, który napisałeś przed chwilą, znajdziesz w następnym rozdziale.

Po ogólnym przedstawieniu pojęcia metod jesteśmy gotowi do omówienia najbardziej obszernego tematu w dziedzinie programowania — klas!



Rysunek 2.7. Skrypt NaukaProgramowania z nową metodą ComputeAge



Rysunek 2.8. Wyjście na konsoli po zmianie wartości zmiennej w oknie Inspector

Klasy — wprowadzenie

Powiedziałem wcześniej, że zmienne przechowują informacje, a metody wykonują działania. Jednak same zmienne i metody to za mało. Potrzebny jest sposób stworzenia czegoś w rodzaju superkontenera. Taki superkontener miałby swoje zmienne i metody, do których można by się odwoływać z wewnątrz. Potrzebujemy klas:

- *Pojęciowo* — klasa przechowuje w pojedynczym kontenerze powiązane ze sobą informacje, działania i zachowania. Klasy mogą nawet komunikować się ze sobą.

- *Technicznie* — klasy są strukturami danych. Mogą zawierać zmienne, metody i inne informacje programowe. Do nich wszystkich można się odwoływać po utworzeniu obiektu klasy.
- *Praktycznie* — klasa jest planem obiektu. Określa reguły i sposób działania dowolnego obiektu (zwanego egzemplarzem) utworzonego za pomocą tego planu.

Prawdopodobnie uświadomiłeś sobie, że klasy otaczają nas nie tylko w Unity, ale także w świecie rzeczywistym. W dalszej części tego rozdziału zajmę się najczęściej używanymi w Unity klasami oraz opisem sposobu ich działania.

Szczegółowy przewodnik po klasach można znaleźć w dokumentacji Microsoftu dotyczącej języka C# pod adresem *https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/classes.*

Popularne klasy w Unity

Zanim zaczniesz się zastanawiać, jak wygląda klasa w C#, powinieneś zdać sobie sprawę, że korzystasz z klasy od początku tego rozdziału. Domyślnie każdy skrypt utworzony w Unity jest klasą. Można to rozpoznać po słowie kluczowym `class` w *wierszu* 3.:

```
public class NaukaProgramowania: MonoBehaviour
{

}
```

`MonoBehaviour` oznacza, że tę klasę można dołączyć do obiektu `GameObject` na scenie Unity. Z kolei dwa nawiasy klamrowe oznaczają granice klasy — do tej klasy należy dowolny kod wewnątrz tych nawiasów.

W C# klasy mogą istnieć samodzielnie. Przekonasz się o tym, gdy zaczniesz samodzielnie tworzyć klasy w rozdziale 5.

Ważna uwaga

Pojęcia skrypt i klasa w kontekście zasobów Unity są czasami używane zamiennie. W celu zachowania spójności będę nazywał pliki C# skryptami, jeśli będą dołączane do kolekcji obiektów GameObjects, oraz klasami, jeśli będą samodzielne.

Klasa jest planem obiektów

W ramach ostatniego przykładu pomyślmy o lokalnym urzędzie pocztowym. Jest to odrębne, samodzielne środowisko, które ma właściwości takie jak fizyczny adres (zmienna) oraz zdolność do wykonywania działań, takich jak wysyłanie kuponu Twojego tajnego dekodera (metoda).

To sprawia, że urząd pocztowy jest doskonałym przykładem potencjalnej klasy, którą możemy naszkicować za pomocą następującego bloku pseudokodu:

```
public class PostOffice {
    // Zmienne
    public string Address = "Dr Otwieracz listów 1234"

    // Metody
    DeliverMail() {}
    SendMail() {}
}
```

Najważniejsze do zapamiętania w przypadku klas jest to, że jeśli informacje i zachowania są zgodne z predefiniowanym projektem, to możliwe jest wykonywanie złożonych działań oraz realizacja komunikacji pomiędzy klasami. Na przykład, gdybyśmy mieli inną klasę, która chciałaby wysłać list za pośrednictwem klasy `PostOffice`, nie musielibyśmy się zastanawiać, co zrobić, by wykonać tę operację. Moglibyśmy utworzyć instancję klasy `PostOffice` i zapisać ją w jej własnej unikalnej zmiennej, aby można było uzyskać do niej dostęp w dowolnym momencie (ponieważ w Twoim mieście może być kilka urzędów pocztowych, trzeba je odróżnić):

```
MyLocalPostOffice = new PostOffice()
```

Można teraz wywołać funkcję `SendMail` instancji klasy `PostOffice` w następujący sposób:

```
MyLocalPostOffice().SendMail()
```

Moglibyśmy także użyć tej klasy do ustalenia adresu urzędu pocztowego. Dzięki temu moglibyśmy ustalić, gdzie należy kierować nasze listy:

```
MyLocalPostOffice().Address
```

Jeśli zastanawiasz się nad wykorzystaniem kropek (tzw. **notacji z kropką**) pomiędzy słowami, proszę o trochę cierpliwości — zajmę się tym pod koniec rozdziału.

Komunikowanie się klas pomiędzy sobą

Do tej pory zajmowaliśmy się klasami lub, szerzej, komponentami Unity jako oddzielnymi, niezależnymi podmiotami. W rzeczywistości jednak są one ze sobą mocno „powiązane". Bez wykorzystania pewnego rodzaju interakcji lub komunikacji pomiędzy klasami trudno byłoby stworzyć jakąkolwiek sensowną aplikację.

W zaprezentowanym wcześniej przykładowym kodzie obsługi urzędu pocztowego zostały użyte kropki do oddzielania od siebie klas, zmiennych i metod. Gdybyśmy pomyśleli o klasach jak o katalogach informacji, to notacja z kropką byłaby narzędziem indeksowania.

Notacja z kropką pozwala na odwoływanie się do zmiennych, metod lub innych typów danych wewnątrz klasy. Dotyczy to również danych zagnieżdżonych lub podklas. Zagadnienie to opiszę dokładniej w rozdziale 5.

Notacja z kropką opisuje również komunikację między klasami. Jest ona używana za każdym razem, gdy klasa potrzebuje informacji o innej klasie lub chce wykonać jedną z jej metod:

```
MyLocalPostOffice().DeliverMail()
```

Notację z kropką czasami określa się jako korzystanie z operatora w postaci kropki (.). Nie zdziw się, jeśli spotkasz się z takim określeniem w dokumentacji.

Jeśli notacja z kropką jeszcze nie jest dla Ciebie jasna, nie martw się, z czasem się do niej przyzwyczaisz. Jest krwioobiegiem całego programowania, pozwala przekazywać informacje i kontekst, gdziekolwiek jest to potrzebne.

Teraz, gdy wiesz już trochę więcej o klasach, porozmawiajmy o narzędziu, którego będziesz najczęściej używać w swojej karierze programistycznej — komentarzach!

Stosowanie komentarzy

Być może zauważyłeś, że w skrypcie *NaukaProgramowania* są dwa dziwne wiersze szarego tekstu zaczynające się od dwóch lewych ukośników (//). Wiersze te zostały utworzone domyślnie podczas tworzenia skryptu.

To są komentarze do kodu. W języku C# jest kilka sposobów tworzenia komentarzy. Środowisko Visual Studio Code (oraz inne aplikacje do edycji kodu) dzięki wbudowanym skrótom często jeszcze bardziej ułatwia ich tworzenie.

Niektórzy profesjonaliści nie nazwaliby komentarzy niezbędnym blokiem budulcowym programowania, ale choć szanuję tę opinię, to jednak się z nią nie zgadzam. Prawidłowe komentowanie kodu za pomocą opisowych informacji jest jednym z najbardziej podstawowych nawyków, jakie powinien mieć nowoczesny programista.

Komentarze jednowierszowe

Poniższy jednowierszowy komentarz jest podobny do tego, który umieściliśmy w skrypcie *NaukaProgramowania*:

```
//To jest komentarz jednowierszowy
```

Visual Studio Code nie kompiluje wierszy zaczynających się od dwóch ukośników (bez pustej spacji) jako kodu, więc możesz ich używać do objaśniania swojego kodu tak często, jak to konieczne. Takie wyjaśnienia mogą się okazać przydatne dla innych osób czytających Twój kod lub dla Ciebie, gdy sięgniesz do niego w przyszłości.

Komentarze wielowierszowe

Zgodnie z nazwą można śmiało założyć, że jednowierszowe komentarze dotyczą tylko jednego wiersza kodu. Jeśli interesują Cię komentarze wielowierszowe, musisz użyć lewego ukośnika i gwiazdki (/*) jako symbolu otwarcia komentarza oraz gwiazdki i lewego ukośnika (*/) jako znacznika zamknięcia komentarza:

```
/*  To jest
        komentarz wielowierszowy  */
```

Możesz także komentować i odkomentowywać bloki kodu przez ich zaznaczenie i użycie kombinacji klawiszy: *Cmd+?* w systemie macOS lub *Ctrl+K+C* w systemie Windows.

Visual Studio Code zapewnia również przydatną własność automatycznego generowania komentarzy. Wpisz trzy lewe ukośniki w wierszu poprzedzającym dowolny wiersz

kodu (z deklaracją zmiennej, metody, klasy lub innych konstrukcji). Spowoduje to wstawienie bloku komentarza z podsumowaniem (rysunek 2.9).

Rysunek 2.9. Wielowierszowy komentarz wygenerowany automatycznie dla metody

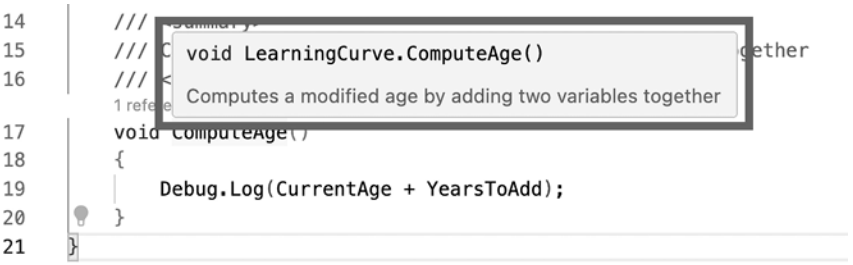
Można oglądać przykładowe komentarze w kodzie, ale żeby zapoznać się z tym mechanizmem, lepiej samodzielnie spróbować je stworzyć. Nigdy nie jest za wcześnie, by zacząć wprowadzać w kodzie komentarze!

Wprowadzanie komentarzy

Otwórz skrypt *NaukaProgramowania* i wprowadź trzy lewe ukośniki nad deklaracją metody `ComputeAge()`.

Powinieneś zobaczyć wygenerowany przez Visual Studio Code wielowierszowy komentarz z opisem metody, umieszczony pomiędzy dwoma znacznikami `<summary>`. Możesz oczywiście zmodyfikować ten tekst lub wprowadzić nowe wiersze. W tym celu wystarczy wcisnąć *Enter*, tak jak w zwykłym dokumencie tekstowym. Pamiętaj, aby nie usunąć znaczników `<summary>`. Jeśli to zrobisz, Visual Studio Code nie będzie w stanie poprawnie rozpoznać komentarzy.

Przeznaczenie tych szczegółowych komentarzy staje się jasne, gdy chcesz się czegoś dowiedzieć o metodzie, którą napisałeś. Jeśli umieściłeś komentarz, zaczynający się od potrójnego lewego ukośnika, i naprowadzisz wskaźnik myszy na nazwę metody w Visual Studio Code, na ekranie wyświetli się podsumowanie zawierające opis metody (rysunek 2.10).



Rysunek 2.10. Wyskakujące okno informacyjne programu Visual Studio z podsumowaniem w formie komentarza

Twój podstawowy zestaw narzędzi programistycznych jest już kompletny (cóż, przynajmniej w teorii). Powinniśmy się teraz zastanowić, jak połączyć wszystkie bloki budulcowe, które poznałeś w tym rozdziale, w środowisku do tworzenia gier Unity. Tym właśnie zajmę się w następnym podrozdziale!

Wykorzystanie bloków budulcowych programowania w praktyce

Po omówieniu bloków budulcowych programowania nadszedł czas, aby przed zakończeniem tego rozdziału wykonać kilka działań w środowisku Unity. W szczególności musisz się dowiedzieć więcej na temat sposobu, w jaki Unity obsługuje skrypty C# dołączone do obiektów `GameObject`.

W tym przykładzie będziemy nadal korzystać ze skryptu *NaukaProgramowania* oraz obiektu `GameObject` *Main Camera*.

Skrypty stają się komponentami

Wszystkie elementy `GameObject` to skrypty. Niektóre z nich napisali dobrzy ludzie z zespołu Unity, inne musimy napisać sami. Komponentów specyficznych dla Unity, takich jak `Transform`, oraz odpowiadających im skryptów po prostu nie powinniśmy edytować.

W chwili gdy utworzony skrypt zostaje upuszczony w kontenerze `GameObject`, staje się kolejnym komponentem tego obiektu, dlatego pojawia się w panelu *Inspector*. Dla Unity ten skrypt „chodzi, mówi i działa" jak każdy inny komponent. „Pod spodem" zawiera zmienne publiczne, które można w dowolnym momencie zmienić. Pomimo że nie powinniśmy edytować komponentów dostarczanych przez Unity, możemy uzyskać dostęp do ich właściwości i metod, dzięki czemu są to potężne narzędzia do programowania.

Gdy skrypt staje się komponentem, Unity wprowadza pewne automatyczne korekty czytelności. Być może zauważyłeś na rysunkach 2.4 i 2.6, że kiedy dodaliśmy do obiektu *Main Camera* skrypt *NaukaProgramowania*, Unity wyświetlił ten skrypt jako *Nauka Programowania*, a zmienną `currentAge` zastąpił nazwą *Current Age*.

W punkcie „Zmienne działają jako symbole zastępcze" przyjrzeliśmy się, jak zaktualizować zmienną w panelu *Inspector*. Warto jednak przyjrzeć się bardziej szczegółowo, jak to działa. Są dwie sytuacje, w których można modyfikować wartość właściwości:

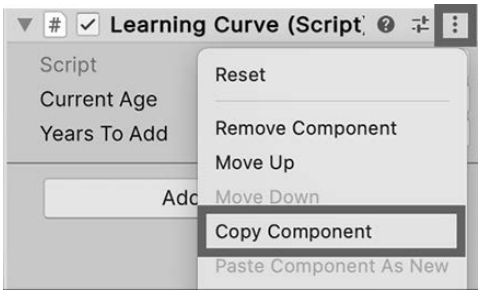
- w trybie odtwarzania, w oknie *Unity Editor* (stan edycji),
- w trybie programowania, w oknie *Unity Editor* (stan wykonywania),
- w edytorze kodu programu Visual Studio Code.

Zmiany wprowadzone w trybie odtwarzania odnoszą skutek natychmiast, w czasie rzeczywistym, co jest świetne podczas testowania i dostrajania gry. Trzeba jednak zapamiętać, że wszelkie zmiany wprowadzone w trybie odtwarzania zostaną utracone, gdy zatrzymamy grę i powrócimy do trybu programowania. Utrata wprowadzonych zmian

w trybie odtwarzania gry może być bardzo frustrująca, zwracaj więc podczas testowania programu baczną uwagę na to, w jakim trybie się znajdujesz.

Aby skopiować wszelkie zmiany wprowadzone w trybie odtwarzania:

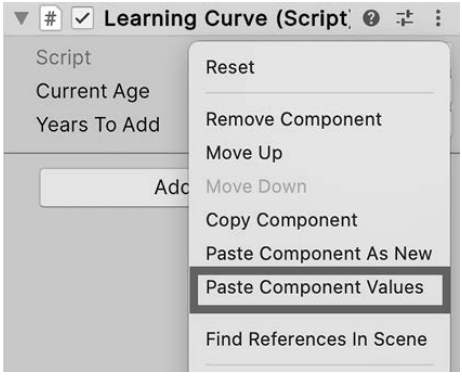
1. Kliknij ikonę z trzema pionowymi kropkami widoczną w prawym górnym rogu komponentu, który zmodyfikowałeś, po czym wybierz polecenie *Copy Component*, tak jak pokazałem na rysunku 2.11.



Rysunek 2.11. Kopiowanie wartości komponentu z panelu Inspector

2. Wyjdź z trybu odtwarzania i ponownie kliknij prawym przyciskiem myszy komponent. Tym razem wybierz polecenie *Paste Component Values* (rysunek 2.12).

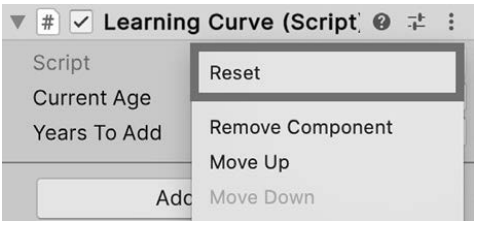
W trybie programowania Unity zapisze wszelkie modyfikacje wprowadzone do zmiennych. Oznacza to, że gdy wyjdziesz z Unity, a następnie uruchomisz program ponownie, zmiany będą zachowane.



Rysunek 2.12. Wklejanie wartości komponentu z panelu Inspector

Zmiany wartości wprowadzone w panelu *Inspector* nie modyfikują skryptu, ale zastępują wszelkie wartości przypisane do skryptu w czasie, gdy środowisko Unity znajdowało się w trybie programowania.

Wszelkie zmiany dokonane w trybie odtwarzania zawsze po jego zatrzymaniu będą automatycznie resetowane. Jeśli chcesz cofnąć zmiany wprowadzone w panelu *Inspector*, możesz zresetować skrypt do jego wartości domyślnych (czasami nazywanych wstępnymi). Kliknij ikonę z trzema kropkami po prawej stronie dowolnego komponentu, a następnie wybierz *Reset*, jak pokazałem na rysunku 2.13.



Rysunek 2.13. Opcja resetowania skryptu w panelu Inspector

Powinno to dać Ci trochę spokoju — jeśli Twoje zmienne wymkną Ci się spod kontroli, zawsze masz do dyspozycji „twardy reset".

Pomocna dłoń od klasy MonoBehaviour

Wiemy, że skrypty C# to klasy, ale skąd Unity wie, aby niektóre ze skryptów przekształcić w komponenty, a inne nie? Krótka odpowiedź jest taka, że *NaukaProgramowania* (a także dowolny inny skrypt stworzony w Unity) dziedziczy po klasie `MonoBehaviour`. Stąd Unity wie, że tę klasę C# można przekształcić na komponent. Jednak nie wszystkie skrypty muszą dziedziczyć po klasie `MonoBehaviour` — jest to konieczne tylko w przypadku tych, które chcemy dodać do obiektów GameObjects w scenach Unity.

Na tym etapie nauki programowania zagadnienie dziedziczenia klas jest nieco zaawansowane. Pomyśl jednak o dziedziczeniu tak, jakby klasa `MonoBehaviour` pożyczyła kilka swoich zmiennych i metod skryptowi *NaukaProgramowania*. Szczegółowe informacje o dziedziczeniu klas znajdziesz w rozdziale 5. Omówię w nim również, jak pisać klasy, które nie dziedziczą po klasie `MonoBehaviour`.

Cykl życia aplikacji Unity

Metody `Start()` i `Update()`, z których skorzystaliśmy w pierwszych przykładach kodu, należą do klasy `MonoBehaviour`. Unity uruchamia te metody automatycznie dla każdego skryptu dodanego do obiektu `GameObject`. Metoda `Start()` uruchamia się raz w momencie rozpoczęcia odtwarzania sceny, a metoda `Update()` uruchamia się raz dla każdej ramki (w zależności od szybkości klatek na określonym komputerze).

Teraz, gdy masz podstawową wiedzę o dokumentacji Unity, mam dla Ciebie do wykonania krótkie, opcjonalne zadanie!

Próba gry Narodziny bohatera — klasa MonoBehaviour w Scripting

Teraz nadszedł czas, abyś nauczył się samodzielnego wykorzystywania dokumentacji Unity. Nie ma na to lepszego sposobu od wypróbowania kilku popularnych metod klasy `MonoBehaviour`.

Aby lepiej zrozumieć rolę metod `Start()` i `Update()` w Unity oraz cele, do jakich są wykorzystywane, spróbuj wyszukać je w dokumentacji *Scripting API*.

Jeśli chcesz, wykonaj dodatkowy krok i poszukaj w dokumentacji opisu klasy `MonoBe↪haviour`. W ten sposób poznasz więcej szczegółowych informacji na jej temat.

Podsumowanie

Na kilku krótkich stronach przebyliśmy długą drogę. Ta lektura pozwoliła jednak zrozumieć teorię podstawowych pojęć, takich jak zmienne, metody i klasy. Powinno to dać Ci solidne podstawy.

Warto pamiętać, że opisane bloki budulcowe mają swoje realistyczne odpowiedniki w prawdziwym świecie. Zmienne przechowują wartości tak, jak skrzynki pocztowe zawierają listy. Metody przechowują instrukcje, przypominające receptury, których należy przestrzegać, aby uzyskać przewidywany produkt, a klasy są projektami przypominającymi realne projekty budynków. Jeśli chcesz, aby budynek przetrwał dłuższy czas, nie da się go zbudować bez dobrze przemyślanego projektu.

W pozostałej części tej książki zagłębimy się w tajniki składni języka C#. Zaczniemy od podstaw. W następnym rozdziale pokażę tylko nieco więcej szczegółów: jak tworzyć zmienne, zarządzać typami wartości oraz korzystać z prostych i bardziej złożonych metod.

Quiz — bloki budulcowe języka C#

a) Jakie jest główne przeznaczenie zmiennej?

b) Jaką rolę w skryptach odgrywają metody?

c) W jaki sposób skrypt staje się komponentem?

d) Jakie jest przeznaczenie notacji z kropką?

Aby zobaczyć, jak Ci poszło, nie zapomnij porównać swoich odpowiedzi z moimi, zamieszczonymi w dodatku „Odpowiedzi na pytania”!

Naucz się C# tam, gdzie liczy się każda linia kodu – w grze!

Programowanie gier w Unity to jedna z najbardziej pożądanych umiejętności na współczesnym rynku IT. Unity pozostaje liderem wśród silników do tworzenia gier — używają go zarówno hobbyści, jak i studia AAA. Kluczem do pełnego wykorzystania jego możliwości jest biegłość w języku C#, który odpowiada za całą logikę i mechanikę rozgrywki. Zapotrzebowanie na programistów Unity stale rośnie, a nauka C# w praktycznym kontekście tworzenia gier to dziś jeden z najskuteczniejszych sposobów wejścia do branży gamedev lub podniesienia kwalifikacji w szeroko pojętym programowaniu obiektowym.

Ta pozycja prowadzi czytelnika od absolutnych podstaw programowania – zmiennych, metod i klas — aż do zaawansowanych zagadnień, takich jak typy generyczne, delegaty, zdarzenia i serializacja danych. Całość jest zilustrowana budową grywalnego prototypu gry akcji Narodziny bohatera w Unity 6. Kolejne rozdziały omawiają mechanikę gracza, sterowanie kamerą, system fizyki, sztuczną inteligencję przeciwników, interfejs użytkownika, zapis i odczyt danych, a także żądania sieciowe. Ósme wydanie tej bestsellerowej serii uwzględnia najnowsze funkcje Unity 6 i stanowi kompletny przewodnik dla każdego, kto chce się nauczyć C# w praktyce.

W książce między innymi:

- Podstawy C# i programowania obiektowego (OOP): zmienne, metody, klasy, dziedziczenie, interfejsy
- Tworzenie grywalnego prototypu 3D w Unity 6 krok po kroku
- Mechanika gracza: ruch, skoki, strzelanie, kolizje i system fizyki Rigidbody
- Sztuczna inteligencja przeciwników z wykorzystaniem NavMesh i agentów nawigacyjnych
- Zapis, odczyt i serializacja danych w formatach: tekstowym, XML i JSON
- Żądania sieciowe i pobieranie danych w czasie rzeczywistym z zewnętrznych API
- Zaawansowane techniki C#: typy generyczne, delegaty, zdarzenia i obsługa wyjątków

Harrison Ferrone — programista i edukator z doświadczeniem zdobytym w Microsoft, PricewaterhouseCoopers i startupach technologicznych. Tworzy kursy dla LinkedIn Learning i specjalizuje się w dokumentacji technicznej. Absolwent Colorado State University i Columbia College Chicago. Pasjonat dostępnej edukacji programistycznej.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-4178-6
9 788328 941786
Cena: 119,00 zł
packt