O'REILLY®

Wydanie IV

Head First C#

Rusz głową!

Doskonały podręcznik
do nauki programowania
w C# i .NET Core

Andrew Stellman
Jennifer Greene

Helion

Tytuł oryginału: Head First C#: A Learner's Guide to Real-World Programming with C# and .NET Core, 4th Edition

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-8322-749-8

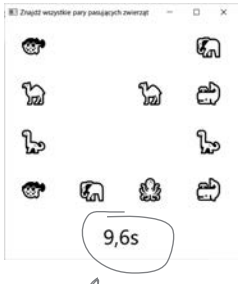
Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/cshr4v
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/cshru4.zip

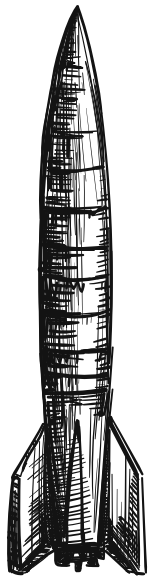
Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści (skrócony)



Spraw, by gra stała się ciekawsza! W dolnej części okna ma się pojawiać czas od rozpoczęcia gry. Czas ma stale biec naprzód, a zatrzymywać się dopiero po dopasowaniu ostatniego zwierzęcia.



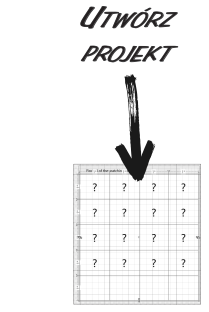
Spis treści (z prawdziwego zdarzenia)

Wprowadzenie

C# według Twojego mózgu. Siedzisz i próbujesz się czegoś *nauczyć*, ale *mózg* stale Ci powtarza, że cała ta nauka *nie jest ważna*. Twój umysł mówi: „Lepiej wyjdź z pokoju i zajmij się ważniejszymi sprawami — dowiedz się, których dzikich zwierząt unikać, a także sprawdź, czy strzelanie z łuku nago to dobry pomysł”. W jaki sposób *możesz* oszukać mózg, aby uznał, że Twoje życie naprawdę zależy od nauki C#?

Zacznij pisać programy w języku C#

1 Utwórz coś wspaniałego... i to szybko



Chcesz tworzyć fantastyczne aplikacje... od razu?

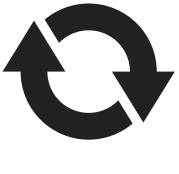
C# to **nowoczesny język programowania** i **cenne narzędzie**, które masz na wyciągnięcie ręki. Ponadto w postaci **Visual Studio** otrzymujesz **fantastyczne środowisko programistyczne** z wysoce intuicyjnymi funkcjami, które sprawiają, że pisanie kodu jest tak łatwe, jak to możliwe. Visual Studio jest nie tylko znakomitym środowiskiem do pisania kodu, ale też **bardzo przydatnym narzędziem do nauki** i eksplorowania języka C#. Brzmi nieźle? Odwróć stronę i zabieraj się do programowania.

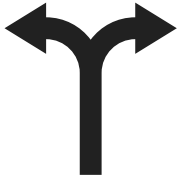
Zanurz się w C#

2 Instrukcje, klasy i kod

Nie jesteś jedynie użytkownikiem IDE. Jesteś programistą.

Za pomocą IDE możesz wykonać wiele zadań, ale nawet to środowisko nie zrobi wszystkiego za Ciebie. Visual Studio jest jednym z najbardziej zaawansowanych narzędzi programistycznych, jakie kiedykolwiek stworzono, ale **rozbudowane IDE** to tylko punkt wyjścia. Pora **zagłębić się w kod C#** i dowiedzieć się, jaką ma budowę, jak działa i jak można przejąć nad nim kontrolę. Nie ma ograniczeń w tym, co mogą robić Twoje aplikacje.

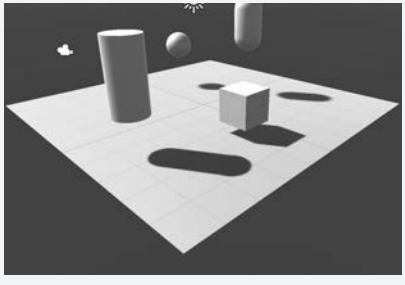


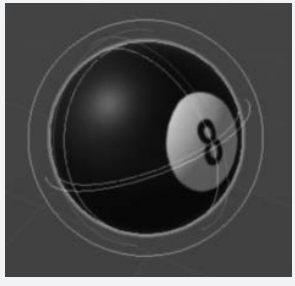


Ćwiczenia z Unity nr 1

Poznawaj C# z Unity

Witaj w pierwszych **ćwiczeniach z Unity w *C#. Rusz głową.*** Pisanie kodu jest umiejętnością i, podobnie jak inne umiejętności, wymaga **praktyki i eksperymentowania**, jeśli chcesz robić postępy. Unity jest wartościowym narzędziem, które Ci w tym pomoże.



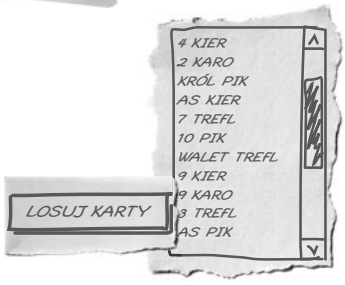


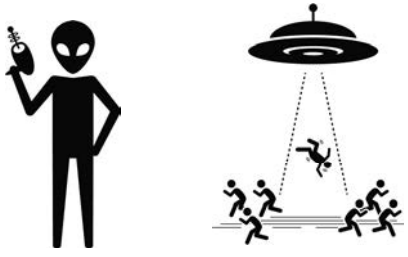
Obiekty — orientuj się!

3 Aby kod miał sens

Każdy program, jaki piszesz, ma rozwiązywać problem.

Gdy tworzysz program, zawsze warto najpierw się zastanowić, jaki *problem* ma rozwiązywać. To dlatego **obiekty** są tak przydatne. Umożliwiają ustrukturyzowanie kodu zgodnie z rozwiązywanym problemem, dzięki czemu możesz *skupić się na nim*, a nie na zawiłościach pisania kodu. Gdy korzystasz z obiektów we właściwy sposób i poświęcasz odpowiednią ilość czasu na ich zaprojektowanie, otrzymujesz kod *prosty* do napisania, a także łatwy do zrozumienia i modyfikowania.



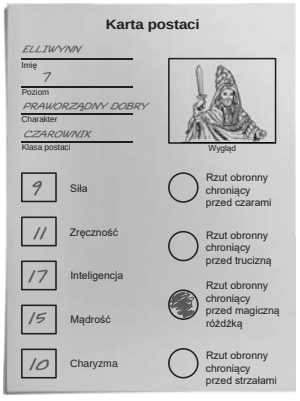


Typy i referencje

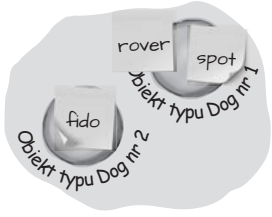
4 Pobieranie referencji

Czym byłyby aplikacje bez danych?

Zastanów się nad tym przez chwilę. Bez danych programy są... no cóż, trudno nawet wyobrazić sobie pisanie kodu bez danych. Potrzebujesz **informacji** od użytkownika i korzystasz z nich do generowania nowych informacji do zwrócenia. Prawie wszystko, co robisz w trakcie programowania, w jakiś sposób dotyczy **pracy z danymi**. W tym rozdziale poznasz tajniki **typów danych** i **referencji** w języku C#, zobaczysz, jak pracować z danymi w programach, a także dowiesz się kilku nowych rzeczy na temat **obiektów** (*i wiesz co — obiekty także są danymi!*).



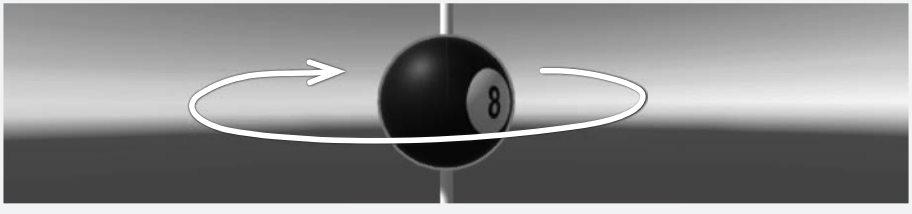
Utworzenie referencji przypomina zapisanie nazwy na karteczce samoprzylepnej i przyklejenie jej do obiektu. Referencji używasz do nazywania obiektu, aby móc go później wskazać.



Ćwiczenia z Unity nr 2

Pisanie kodu C# w Unity

Unity nie jest *tylko* rozbudowanym, międzyplatformowym silnikiem i edytorem do budowania gier 2D i 3D oraz symulacji. Jest też **doskonałym narzędziem do nabierania praktyki w pisaniu kodu C#**. W tych ćwiczeniach nabierzesz wprawy w pisaniu kodu C# w projekcie w Unity.

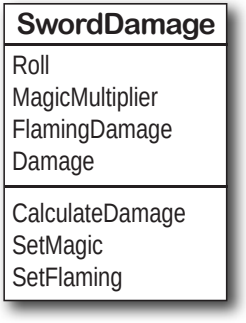


Hermetyzacja

5 Zapewnij prywatność swoich sekretów

Czy marzyłeś kiedyś o odrobinie prywatności?

Obiekty czasem czują się podobnie. Tak jak Ty nie chcesz, aby osoby, którym nie ufasz, czytały Twój dziennik lub przeglądały wyciągi bankowe, dobre obiekty nie pozwalają ***innym*** obiektom podglądać swoich pól. W tym rozdziale poznasz wartość **hermetyzacji**. Jest to mechanizm programistyczny pomagający pisać kod łatwy do modyfikowania i stosowania oraz odporny na próby niewłaściwego użycia. **Zapewnisz prywatność danym obiektów** oraz dodasz **właściwości** zabezpieczające dostęp do tych danych.



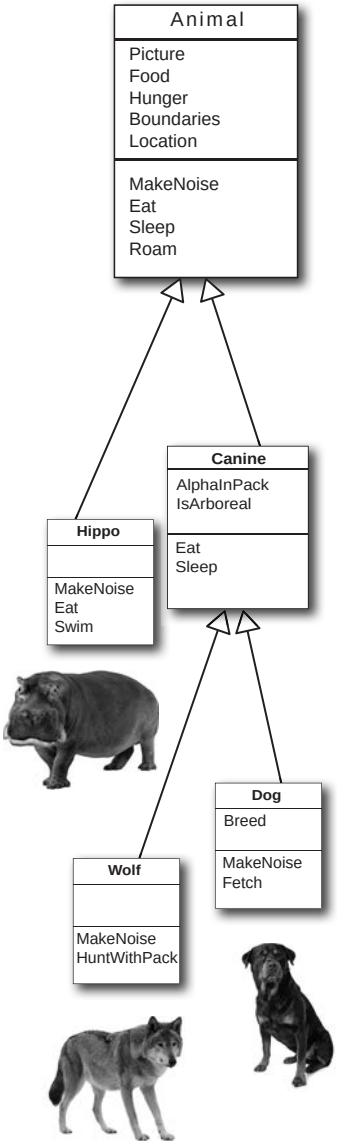

```
RealName: „Henryk Janik"
Alias: „Damian Marczuk"
Password: „kruk lata o północy"
```

Dziedziczenie

6 Drzewo genealogiczne obiektów

Czasem *CHCESZ* być taki jak Twoi rodzice.

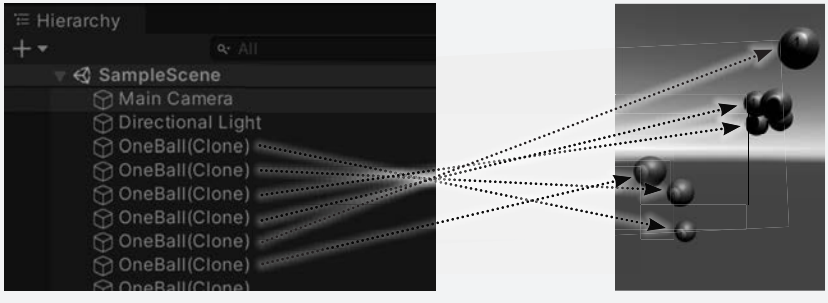
Czy natrafiłeś kiedyś na klasę, która robi ***prawie*** dokładnie to, co ma robić ***Twoja*** klasa? Czy naszła Cię myśl, że gdybyś tylko mógł ***zmienić kilka rzeczy***, klasa byłaby idealna? Dzięki **dziedziczeniu** możesz **rozszerzać** istniejące klasy. W ten sposób nowa klasa otrzyma wszystkie operacje starej, zachowując **swobodę** wprowadzania zmian, a Ty będziesz mógł dostosować nową klasę do swoich potrzeb. Dziedziczenie jest jedną z najważniejszych koncepcji i technik w języku C#. Dzięki niemu możesz **uniknąć powtórzeń w kodzie**, uzyskać dokładniejszy **model rzeczywistego świata** i budować aplikacje, które są **łatwiejsze w konserwacji** i **mniej narażone na błędy**.



Ćwiczenia z Unity nr 3

Instancje obiektów gry

C# jest językiem obiektowym, a ponieważ celem ćwiczeń z Unity w książce *Rusz głową. C#* jest *praktykowanie pisania kodu w C#*, zrozumiałe jest, że w tych ćwiczeniach skupisz się na tworzeniu obiektów.

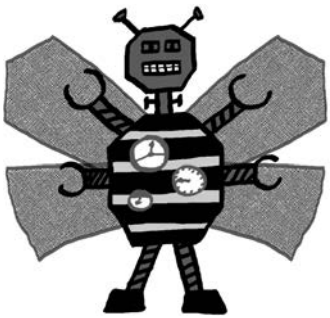


Interfejsy, rzutowanie i instrukcja „is”

7 Spełnianie obietnic przez klasy

Chcesz, aby obiekt wykonywał określone zadanie? Zastosuj interfejs.

Czasem programista chce pogrupować obiekty na podstawie **tego, co potrafią robić**, a nie na bazie klas, po jakich dziedziczą. **Interfejsy** to umożliwiają. Możesz użyć interfejsu do zdefiniowania **określonego zadania**. Każda instancja klasy **implementującej** dany interfejs *gwarantuje wykonanie tego zadania* niezależnie od tego, po jakich klasach dziedziczy. Aby ten mechanizm działał, każda klasa implementująca interfejs musi zobowiązać się do **spełnienia wszystkich obietnic** — w przeciwnym razie kompilator połamie jej kolana, rozumiesz?



Wyliczenia i kolekcje

8 Porządkowanie danych

Dane nie zawsze są tak uporządkowane, jak byś sobie tego życzył.

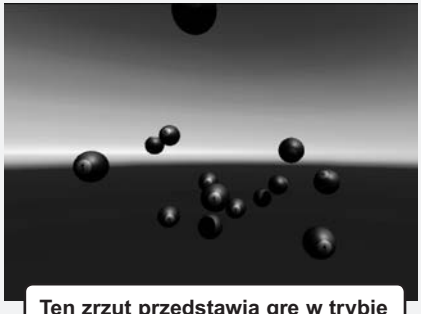
W praktyce nie otrzymujesz danych w postaci uporządkowanych porcji i elementów. Nic z tego — dane będą trafiać do Ciebie w postaci rozmaitych **ładunków, stert i wiązek**. Będziesz potrzebować rozbudowanych narzędzi, aby je wszystkie uporządkować. Przydadzą Ci się do tego **wyliczenia** i **kolekcje**. Wyliczenia to typy, które pozwalają zdefiniować poprawne wartości do kategoryzowania danych. Kolekcje są specjalnymi obiektami przechowującymi wiele wartości. Pozwalają **przechowywać i sortować** wszelkie dane, jakie program musi przetwarzać, oraz **zarządzać** nimi. Dzięki temu możesz przeznaczyć czas na zastanowienie się nad programem pracującym z danymi, a obsługą ich samych zajmą się kolekcje.

Rzadko używana karta „książę bawołów".

Ćwiczenia z Unity nr 4

Interfejsy użytkownika

W poprzednich ćwiczeniach z Unity zacząłeś pisać grę, używając obiektów prefab do tworzenia obiektów gry, które pojawiają się w losowych punktach przestrzeni gry trójwymiarowej i latają po ekranie. W tych ćwiczeniach z Unity będziesz kontynuować te prace, co pozwoli Ci wykorzystać wiedzę na temat interfejsów z C# i inne informacje.



Ten zrzut przedstawia grę w trybie rozgrywki. Bile są dodawane, a gracz może je klikać, aby poprawić wynik.

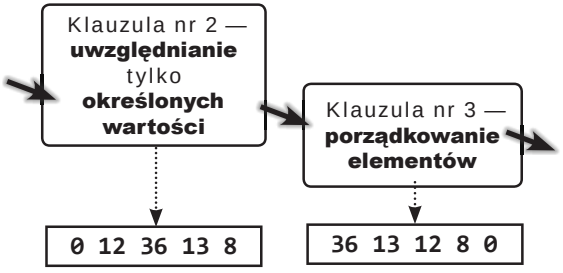
Po dodaniu ostatniej bili gra przechodzi w tryb „Koniec gry". Wyświetlany jest przycisk „Zagraj ponownie", a gra nie dodaje nowych bil.

LINQ i lambdy

9 Kontroluj swoje dane

Świat jest sterowany przez dane. Wszyscy musimy nauczyć się w nim żyć.

Minęły już dni, kiedy można było dniami, a nawet tygodniami programować bez przetwarzania **dużych ilości danych**. Dziś ***wszystko kręci się wokół danych***. Tu właśnie przydaje się technologia **LINQ**. Jest to mechanizm języka C# i platformy .NET, który nie tylko pozwala w intuicyjny sposób **pisać kwerendy danych** z kolekcji platformy .NET, ale też **grupować dane** i **scalać dane z różnych źródeł**. W tym rozdziale dodasz **testy jednostkowe**, aby mieć pewność, że kod działa w oczekiwany sposób. Gdy już nauczysz się przekształcać dane na możliwe do przetworzenia porcje, zastosujesz **wyrażenia lambda**, aby zrefaktoryzować kod C#, by był jeszcze bardziej zwięzły.

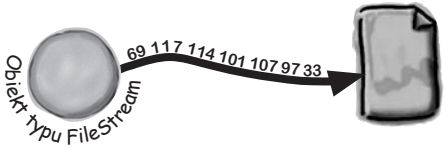


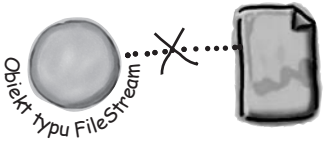
Odczyt i zapis plików

10 Zachowaj dla mnie ostatni bajt

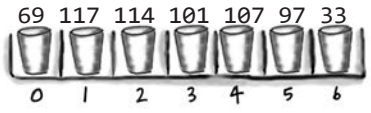
Czasem opłaca się być (wy)trwałym.

Do tej pory wszystkie omawiane programy miały krótki czas życia. Były uruchamiane, działały przez pewien czas, po czym kończyły pracę. Jednak ten model nie zawsze wystarcza — zwłaszcza gdy przetwarzasz ważne informacje. Potrzebna jest możliwość **zapisywania pracy**. W tym rozdziale zobaczysz, jak **zapisywać dane w pliku**, a także jak **wczytywać informacje** z pliku. Poznasz też **strumienie**, dowiesz się, jak zapisywać obiekty w plikach z użyciem **serializacji**, i przyjrzysz się bitom i bajtom w **danych szesnastkowych, Unicode i dwójkowych**.





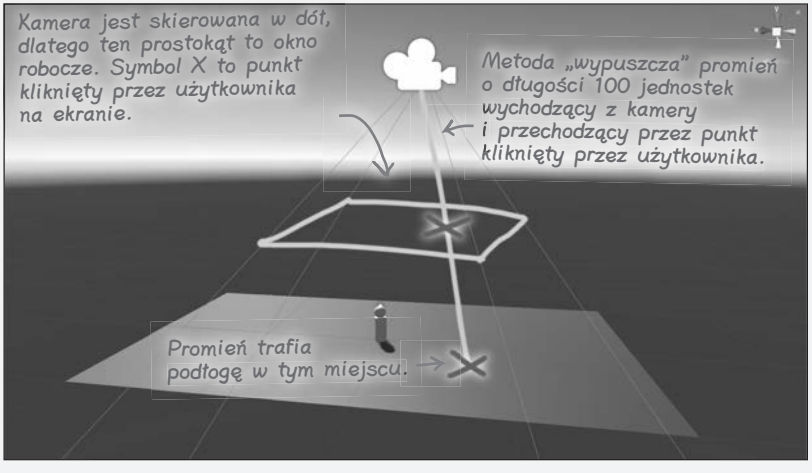
Eureka! →



Ćwiczenia z Unity nr 5

Ray casting

Gdy konfigurujesz scenę w Unity, tworzysz wirtualny trójwymiarowy świat, w którym postacie z gry mogą się poruszać. Jednak w większości gier wiele wydarzeń nie jest bezpośrednio kontrolowanych przez gracza. W jaki więc sposób obiekty poruszają się po scenie? W tych ćwiczeniach zobaczysz, jak C# może w tym pomóc.



KAPITAN WSPANIAŁY

ŚMIERĆ OBIEKTU

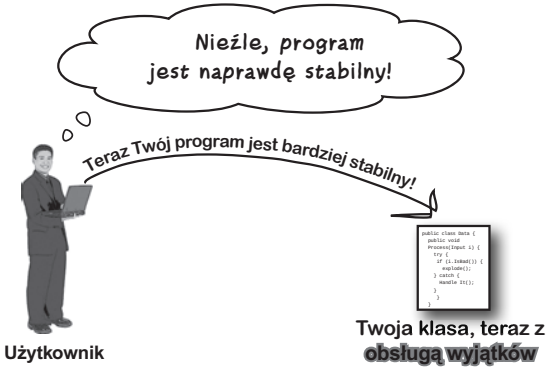
C#. Rusz głową	
4 zł	Rozdział 11.

Obsługa wyjątków

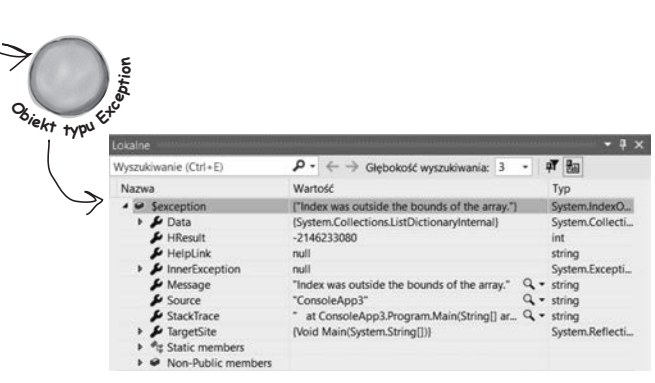
12 Gaszenie pożarów staje się nudne

Programiści nie powinni być strażakami.

Ciężko pracowałeś, przedzierając się przez podręczniki techniczne i kilka wciągających książek z serii *Rusz głową*, i dotarłeś na sam szczyt swojej profesji. Ale spanikowani współpracownicy nadal dzwonią do Ciebie w środku nocy, ponieważ **Twój program się zawiesza** lub **nie działa tak, jak powinien**. Nic nie wybija programistów z rytmu tak, jak konieczność wyeliminowania dziwnego błędu. Jednak dzięki **obsłudze wyjątków** możesz pisać kod **radzący sobie z problemami**, które się pojawiają. Jeszcze lepsze jest to, że możesz nawet zaplanować działania na wypadek takich problemów, aby program **kontynuował działanie** po ich wystąpieniu.



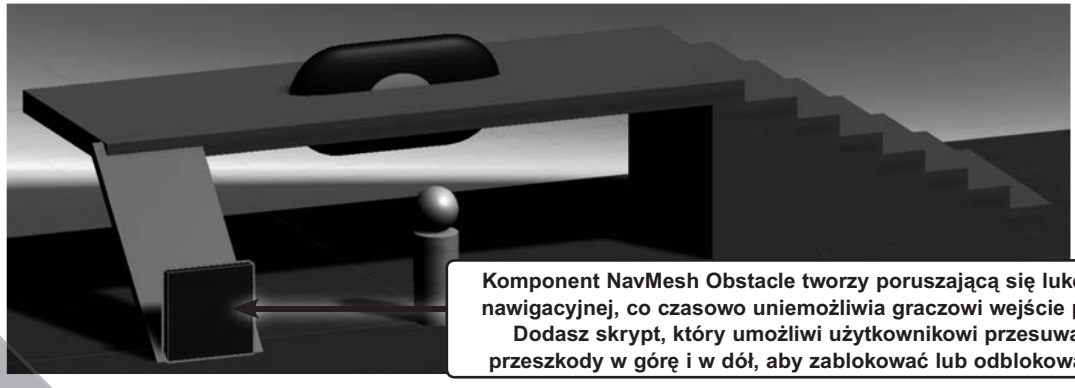

```
int[] anArray = {3, 4, 1, 11};
int aValue = anArray[15];
```



Ćwiczenia z Unity nr 6

Nawigowanie po scenie

W poprzednich ćwiczeniach z Unity utworzyłeś scenę z podłogą (płaszczyzną) i gracza (kulę powiązaną z walcem). Ponadto użyłeś siatki nawigacyjnej, obiektu NavMesh Agent i ray castingu, aby gracz podążał w miejsca kliknięcia myszą na scenie. W tych ćwiczeniach dodasz elementy do sceny, używając C#.



Komponent NavMesh Obstacle tworzy poruszającą się lukę w siatce nawigacyjnej, co czasowo uniemożliwia graczowi wejście po rampie. Dodasz skrypt, który umożliwi użytkownikowi przesuwanie tej przeszkody w górę i w dół, aby zablokować lub odblokować rampę.

Projekty ASP.NET Core Blazor

A

Przewodnik ucznia po Visual Studio for Mac

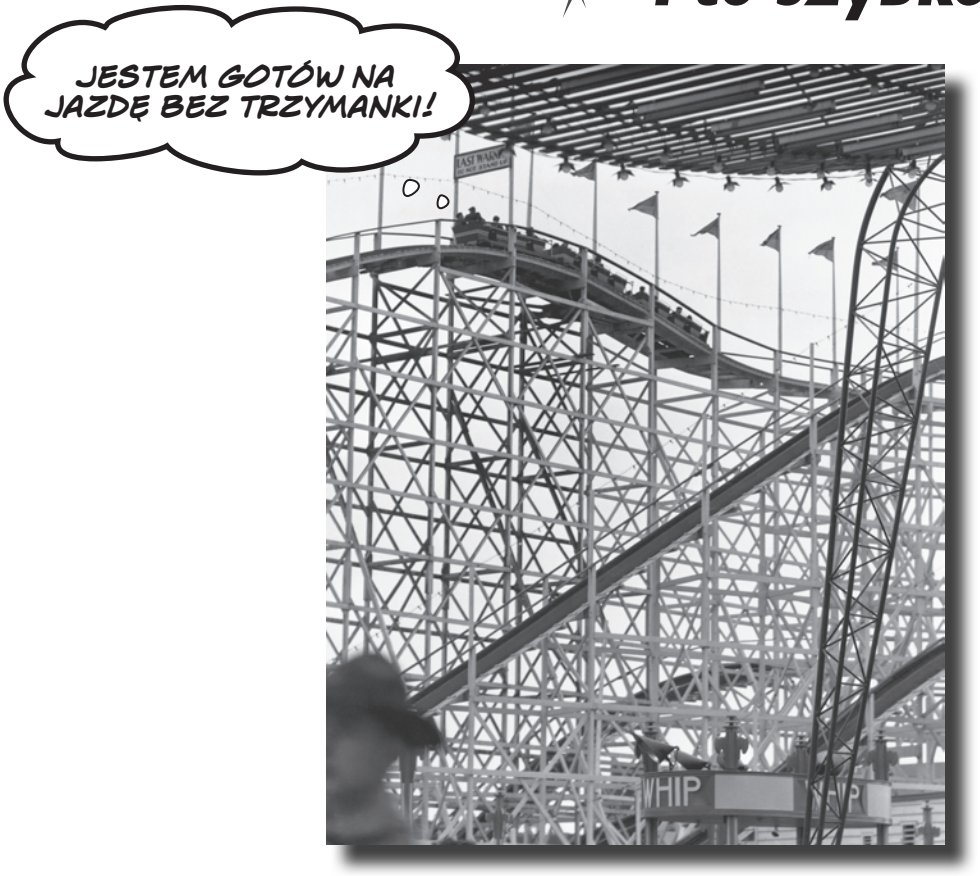
Twój Mac jest pełnoprawnym obywatelem w świecie C# i .NET.

Kata z programowania

B

Podręcznik dla zaawansowanych i/lub niecierpliwych

Utwórz coś wspaniałego... i to szybko!



Chcesz tworzyć fantastyczne aplikacje... od razu?
C# to **nowoczesny język programowania** i **cenne narzędzie**, które masz na wyciągnięcie ręki. Ponadto w postaci **Visual Studio** otrzymujesz **fantastyczne środowisko programistyczne** z wysoce intuicyjnymi funkcjami, które sprawiają, że pisanie kodu jest tak łatwe, jak to możliwe. Visual Studio jest nie tylko znakomitym środowiskiem do pisania kodu, ale też **bardzo przydatnym narzędziem do nauki** i eksplorowania języka C#. Brzmi nieźle? Odwróć stronę i zabieraj się do programowania.

Dlaczego warto opanować C#?

C# jest prostym, nowoczesnym językiem, który umożliwia uzyskanie niezwykłych efektów. W trakcie nauki C# poznasz nie tylko sam język. C# otworzy przed Tobą cały świat platformy .NET i da Ci dostęp do otwartej platformy o wielkich możliwościach, która pozwala na pisanie aplikacji różnego rodzaju.

Upewnij się, że instalujesz środowisko Visual Studio, a nie Visual Studio Code.

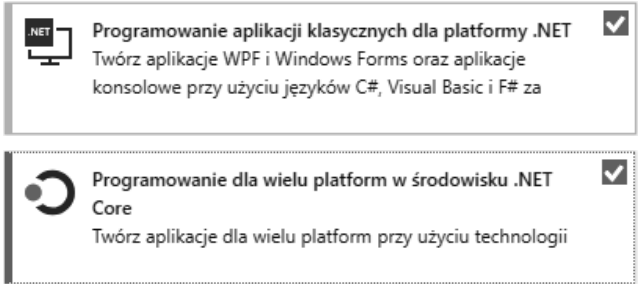
Visual Studio Code jest świetnym otwartym edytorem kodu działającym w różnych systemach, ale nie jest dostosowany do programowania na platformie .NET tak dobrze jak Visual Studio. To dlatego w tej książce to Visual Studio będzie używane jako narzędzie do nauki i poznawania C#.

Visual Studio jest Twoją przepustką do języka C#

Jeśli jeszcze nie zainstalowałeś środowiska Visual Studio 2019, pora to zrobić. Otwórz stronę *visualstudio.microsoft.com* i **pobierz wersję Visual Studio Community**. Jeżeli masz ją już zainstalowaną, uruchom instalator środowiska Visual Studio, aby zaktualizować zainstalowane komponenty.

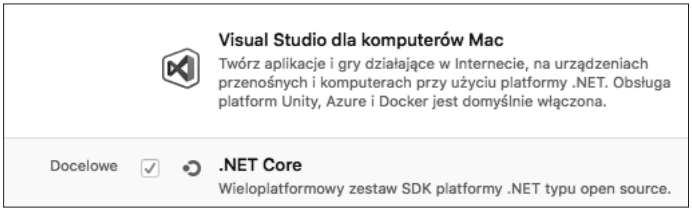
Jeśli używasz systemu Windows...

Koniecznie zaznacz opcje umożliwiające programowanie dla wielu platform w środowisku .NET Core i programowanie aplikacji desktopowych na platformę .NET. Nie zaznaczaj jednak opcji *Opracowywanie gier za pomocą aparatu Unity*. W dalszych rozdziałach książki zajmiesz się programowaniem gier 3D z użyciem silnika Unity, ale zainstalujesz go osobno.



Jeśli używasz systemu Mac...

Pobierz i uruchom instalator środowiska Visual Studio for Mac. Upewnij się, że zaznaczona jest docelowa platforma .NET Core.



W systemie Windows możesz też tworzyć projekty ASP.NET! Wystarczy zaznaczyć opcję „Opracowywanie zawartości dla platformy ASP.NET i sieci Web" w trakcie instalowania środowiska Visual Studio.

Większość aplikacji z tej książki to aplikacje konsolowe na platformę .NET Core, które działają w systemach Windows i Mac. W niektórych rozdziałach opisane są projekty dla systemu Windows, na przykład gra w dopasowywanie zwierząt, o której mowa w dalszej części tego rozdziału. W przypadku takich projektów zajrzyj do dodatku „Przewodnik ucznia po Visual Studio for Mac". Znajdziesz tam *kompletny zastępnik* kodu z rozdziału 1. i wersje innych projektów WPF bazujące na platformie ASP.NET Core Blazor.

Visual Studio jest narzędziem do pisania kodu oraz poznawania języka C#

Do pisania kodu w języku C# możesz używać notatnika lub innego edytora tekstu, jednak istnieje lepsze rozwiązanie. **Zintegrowane środowisko programistyczne** (ang. *integrated development environment* — **IDE**) to edytor tekstu, narzędzie do projektowania wizualnego, menedżer plików, debuger — jest jak scyzoryk szwajcarski do wykonywania wszystkich zadań związanych z pisaniem kodu.

Oto tylko kilka wybranych operacji, w jakich Visual Studio Ci pomoże:

1. **SZYBKIE tworzenie aplikacji.** Język C# ma duże możliwości i jest łatwy w nauce, a IDE Visual Studio upraszcza prace, automatycznie wykonując wiele żmudnych zadań. Oto kilka czynności, które Visual Studio wykonuje za programistę (lub w których bardzo mu pomaga):

   - ★ zarządzanie wszystkimi plikami projektu,
   - ★ ułatwianie edycji kodu projektu,
   - ★ zarządzanie grafiką, plikami dźwiękowymi, ikonami i innymi zasobami projektu,
   - ★ wsparcie w debugowaniu kodu przez wykonywanie go wiersz po wierszu.

2. **Projektowanie atrakcyjnego interfejsu użytkownika.** Narzędzie do projektowania wizualnego w IDE Visual Studio jest jednym z najłatwiejszych w użyciu narzędzi tego typu. Wykonuje za programistę tyle pracy, że przekonasz się, iż tworzenie interfejsów użytkownika będzie jednym z najbardziej satysfakcjonujących aspektów tworzenia aplikacji w C#. Możesz tworzyć kompletne profesjonalne programy bez konieczności poświęcania wielu godzin na dopracowywanie interfejsu użytkownika (chyba że będziesz miał na to ochotę).

Jeśli używasz środowiska Visual Studio for Mac, możesz tworzyć równie atrakcyjnie wyglądające aplikacje, ale zamiast używać języka XAML, połącz C# z HTML-em.

3. **Tworzenie znakomicie wyglądających programów.** Gdy **połączysz C# z XAML-em**, językiem znaczników służącym do projektowania interfejsów użytkownika dla aplikacji desktopowych WPF, uzyskasz jedno z najwydajniejszych narzędzi do tworzenia programów z interfejsem graficznym. Możesz wykorzystać je do tworzenia oprogramowania, które wygląda równie dobrze, jak działa.

Interfejs użytkownika aplikacji WPF jest tworzony w języku XAML (eXtensible Application Markup Language). W Visual Studio praca z językiem XAML jest naprawdę prosta.

4. **Nauka i eksplorowanie języka C# oraz platformy .NET.** Visual Studio jest najwyższej klasy narzędziem programistycznym, ale — na szczęście dla nas — także znakomitym narzędziem wspomagającym naukę. ***Użyjemy tu tego środowiska IDE do eksplorowania języka C#***, co pozwoli *szybko* przenieść ważne zagadnienia programistyczne do Twojego mózgu.

W tej książce Visual Studio czasem będzie nazywane po prostu „środowiskiem IDE".

Visual Studio to fantastyczne środowisko programistyczne, ale tu posłuży także do eksplorowania języka C#.

Utwórz pierwszy projekt w Visual Studio

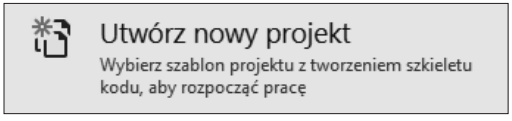
Najlepszy sposób na naukę języka C# polega na rozpoczęciu pisania kodu. Dlatego użyjesz Visual Studio do **utworzenia nowego projektu**… i od razu zaczniesz pisać kod!

Zrób to!

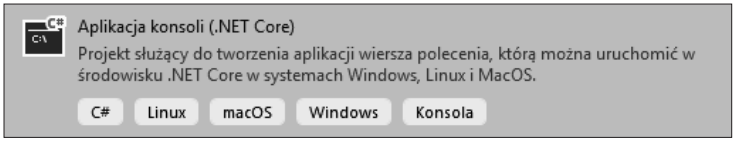
Gdy natrafisz na tekst Zrób to! (lub Zrób to teraz!, Debuguj to! itd.), otwórz Visual Studio i wykonaj opisane kroki. Dokładnie wyjaśniamy, co należy zrobić, i pokazujemy, na co zwrócić uwagę, aby jak najbardziej skorzystać z danego przykładu.

1 Tworzenie nowego projektu dla aplikacji konsolowej na platformę .NET Core.

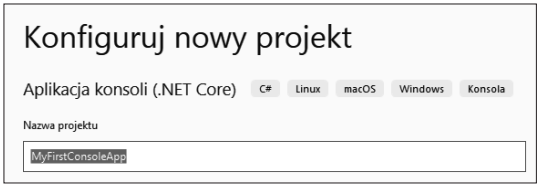
Uruchom środowisko Visual Studio 2019. Po uruchomieniu pojawi się okno *Utwórz nowy projekt* z kilkoma różnymi opcjami. Wybierz opcję ***Utwórz nowy projekt***. Nie martw się, jeśli zamknąłeś okno — zawsze możesz do niego wrócić, wybierając w menu opcję *Plik/Nowy/Projekt*.



Wybierz projekt typu ***Aplikacja konsoli (.NET Core)***, klikając go. Następnie kliknij przycisk ***Dalej***.



Nazwij projekt ***MyFirstConsoleApp*** i kliknij przycisk ***Utwórz***.



*Jeśli używasz **Visual Studio for Mac**, kod tego projektu (i wszystkich projektów aplikacji konsolowych dla .NET Core) będzie wyglądać identycznie, ale niektóre funkcje IDE będą inne. Wersję tego rozdziału dla komputerów Mac znajdziesz w dodatku **„Przewodnik ucznia po Visual Studio for Mac”**.*

2 Przyjrzyj się kodowi nowej aplikacji.

Gdy Visual Studio tworzy nowy projekt, generuje punkt wyjścia, od którego możesz zacząć. Zaraz po zakończeniu tworzenia nowych plików aplikacji środowisko powinno otworzyć plik *Program.cs* z następującym kodem:

```
Odwołania: 0
class Program
{
    Odwołania: 0
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Gdy Visual Studio tworzy nowy projekt aplikacji konsolowej, automatycznie dodaje klasę o nazwie Program.

Ta klasa rozpoczyna się od metody Main, która zawiera jedną instrukcję, wyświetlającą w konsoli wiersz tekstu. Klasy i metody są dokładnie opisane w rozdziale 2.

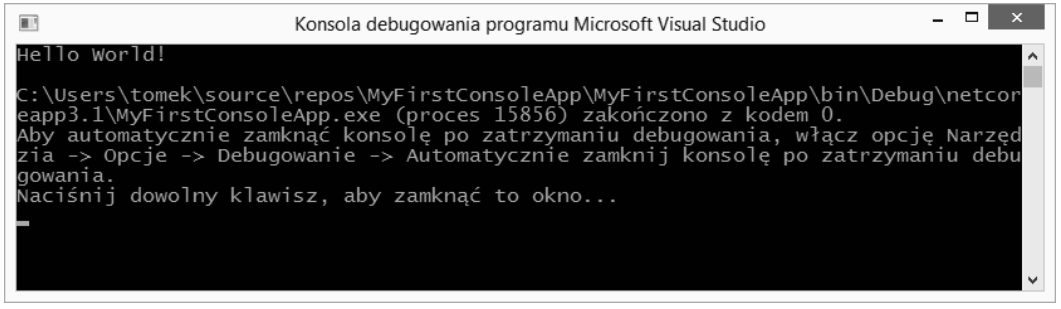
3 Uruchom nową aplikację.

Aplikacja, którą Visual Studio utworzyło za Ciebie, jest gotowa do uruchomienia. W górnej części IDE Visual Studio znajdź przycisk z zielonym trójkątem i nazwą aplikacji, po czym kliknij go:

4 Przyjrzyj się danym wyjściowym programu.

Gdy uruchomisz aplikację, pojawi się okno **Konsola debugowania programu Microsoft Visual Studio** z danymi wyjściowymi programu:

Gdy uruchomisz aplikację, ta wykona metodę Main, która wyświetla ten wiersz tekstu w konsoli.



Najlepszy sposób na naukę języka polega na pisaniu w nim dużej ilości kodu. Dlatego w tej książce będziesz pisać wiele programów. Liczne z nich będą projektami aplikacji konsolowych na platformę .NET Core, warto więc przeanalizować to, co właśnie zrobiłeś.

W górnej części okna znajdują się **dane wyjściowe programu**:

```
Hello World!
```

Dalej znajduje się pusty wiersz, a następnie dodatkowy tekst:

```
C:\<ścieżka-do-katalogu-projektu>\MyFirstConsoleApp\MyFirstConsoleApp\bin\Debug\
netcoreapp3.1\MyFirstConsoleApp.exe (proces #####) zakończono z kodem 0.
Aby automatycznie zamknąć konsolę po zatrzymaniu debugowania, włącz opcję Narzędzia->
opcje->Debugowanie->Automatycznie zamknij konsolę po zatrzymaniu debugowania.
Naciśnij dowolny klawisz, aby zamknąć to okno. . .
```

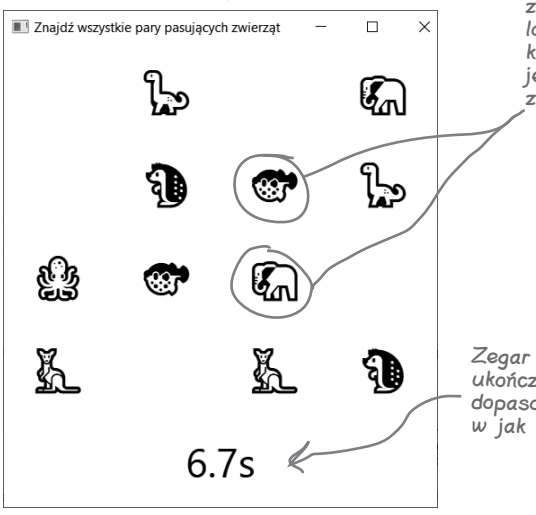
Ten sam komunikat zobaczysz w dolnej części każdego okna Konsoli debugowania. Omawiany program wyświetla wiersz tekstu (**`Hello World!`**), po czym kończy pracę. Visual Studio pozostawia okno otwarte do czasu wciśnięcia klawisza, co powoduje zamknięcie go. Dzięki temu możesz zobaczyć dane wyjściowe przed zniknięciem okna.

Wciśnij klawisz, aby zamknąć okno. Następnie ponownie uruchom program. W ten sposób będziesz uruchamiać wszystkie projekty aplikacji konsolowych na platformę .NET Core, jakie napiszesz w trakcie lektury tej książki.

Napiszmy grę!

Zbudowałeś swoją pierwszą aplikację w C# — świetnie! Teraz pora utworzyć coś trochę bardziej skomplikowanego. Teraz zbudujesz **grę w dopasowywanie zwierząt**, w której gracz widzi zestaw 16 zwierzaków i musi klikać pasujące pary, aby te zniknęły.

Gra w dopasowywanie zwierząt, jaką zbudujesz.



Gra wyświetla osiem par zwierząt rozmieszczonych losowo w oknie. Gracz klika dwa zwierzęta; jeśli te pasują do siebie, znikają z okna.

Zegar odmierza czas ukończenia gry. Celem jest dopasowanie wszystkich par w jak najkrótszym czasie.

Tworzenie projektów różnego rodzaju jest ważnym aspektem nauki języka C#. W niektórych projektach w tej książce używana jest technologia WPF (ang. *Windows Presentation Foundation*), ponieważ zapewnia narzędzia do projektowania zaawansowanych interfejsów użytkownika działających w różnych wersjach systemu Windows (nawet tak dawnych jak Windows XP).

Jednak C# jest przeznaczony nie tylko dla systemu Windows!

Używasz komputerów Mac? Masz szczęście! Specjalnie dla Ciebie dodaliśmy materiały dotyczące Visual Studio for Mac. W dodatku „Przewodnik ucznia po Visual Studio for Mac" na końcu książki znajdziesz kompletny zastępnik tego rozdziału i wersje wszystkich projektów WPF z tej książki dostosowane do komputerów Mac.

W wersjach projektów WPF dla komputerów Mac używana jest technologia ASP.NET Core. Projekty ASP.NET Core możesz budować także dla systemu Windows.

Gra w dopasowywanie zwierząt jest aplikacją WPF

Aplikacje konsolowe świetnie się sprawdzają, jeśli chcesz wprowadzać i wyświetlać sam tekst. Jeżeli potrzebujesz wizualnej aplikacji wyświetlanej w oknie, musisz zastosować inną technologię. To dlatego gra w dopasowywanie zwierząt będzie **aplikacją WPF**. WPF umożliwia tworzenie aplikacji desktopowych na różne wersje systemu Windows. W większości rozdziałów z tej książki opisywana będzie jedna aplikacja WPF. Celem tego projektu jest przedstawienie technologii WPF i zapewnienie Ci narzędzi do tworzenia atrakcyjnych wizualnie aplikacji desktopowych (obok aplikacji konsolowych).

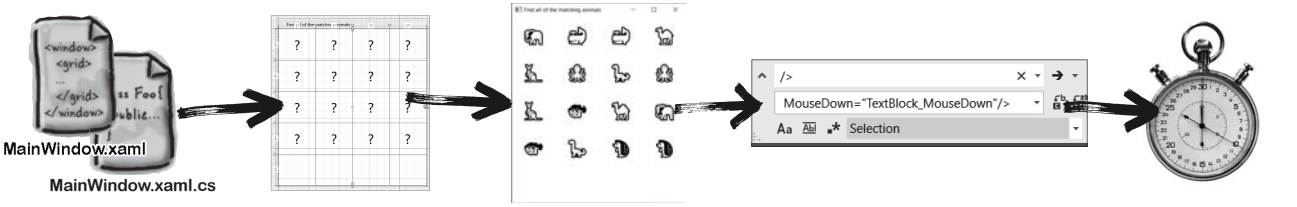
Do czasu zakończenia pracy nad tym projektem dużo lepiej poznasz narzędzia, z których będziesz korzystać w tej książce do uczenia się i eksplorowania języka C#.

Proces tworzenia gry

W dalszej części rozdziału opisany jest proces budowania gry w dopasowywanie zwierząt. Proces ten jest podzielony na kilka odrębnych etapów:

1. Najpierw utworzysz nowy projekt aplikacji desktopowej w Visual Studio.
2. Następnie użyjesz języka XAML do zbudowania okna.
3. Potem napiszesz kod w C#, aby dodać do okna losowe emoji ze zwierzętami.
4. Gra wymaga od użytkownika klikania par pasujących emoji.
5. Na zakończenie trzeba dodać zegar, dzięki czemu gra będzie bardziej ekscytująca.

Realizacja tego projektu może zająć od 15 minut do godziny (w zależności od tego, jak szybko piszesz kod). Nauka daje lepsze efekty, gdy się nie spieszysz, dlatego zapewnij sobie odpowiednio dużo czasu.



Utwórz projekt → *Zaprojektuj okno* → *Napisz kod w C#* → *Dodaj obsługę kliknięć myszą* → *Dodaj zegar*

Zwróć uwagę na zamieszczone w książce ramki „Projektowanie gier... i nie tylko". Zasady projektowania gier posłużą do nauki i eksploracji ważnych koncepcji i zagadnień z obszaru programowania, które mają zastosowanie w każdym projekcie — nie tylko w grach komputerowych.

Projektowanie gier... i nie tylko

Czym jest gra?

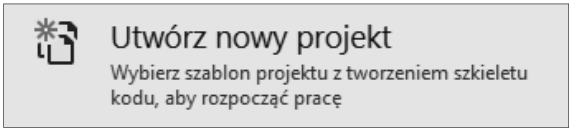
Może się wydawać, że oczywiste jest, czym jest gra. Zastanów się jednak nad tym przez chwilę — sprawa wcale nie jest taka prosta.

- Czy wszystkie gry mają **zwycięzcę**? Czy zawsze się kończą? Niekoniecznie. Jak to wygląda w symulatorze lotu, w grze w projektowanie parku rozrywki lub w grach takich jak The Sims?
- Czy gry zawsze są **ciekawe**? Nie dla każdego. Niektórzy gracze lubią „grind", czyli powtarzanie w kółko tych samych czynności. Inni nie cierpią takich gier.
- Czy w grach zawsze występuje **podejmowanie decyzji, konflikt lub rozwiązywanie problemów**? Nie we wszystkich. W grach typu *walking simulator* (dosłownie „symulator chodzenia") gracz jedynie eksploruje otoczenie. Często nie ma w nich żadnych zagadek ani konfliktów.
- Okazuje się, że dość trudno jest precyzyjnie określić, czym jest gra. Jeśli zajrzysz do podręczników projektowania gier, znajdziesz rozmaite definicje. Dlatego na potrzeby tej książki zdefiniujmy **znaczenie słowa „gra"** tak:

Gra jest programem, którego używanie daje (miejmy nadzieję) przynajmniej tyle przyjemności, co jego pisanie.

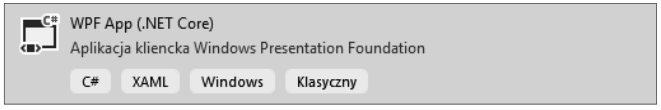
Utwórz projekt WPF w Visual Studio

Uruchom nową instancję środowiska Visual Studio 2019 i utwórz nowy projekt:

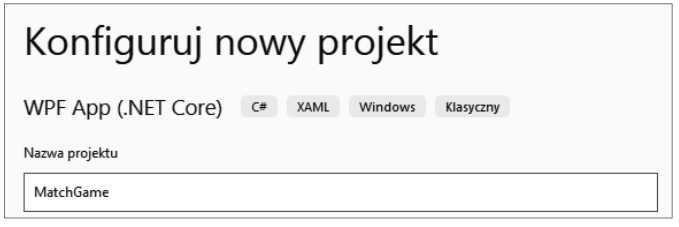


Skończyłeś już pracę nad projektem aplikacji konsolowej z pierwszej części rozdziału, dlatego możesz zamknąć instancję środowiska Visual Studio z tamtym programem.

Grę zbudujesz jako aplikację desktopową za pomocą technologii WPF. Wybierz więc **WPF App (.NET Core)** i kliknij *Dalej*:



Visual Studio wyświetli prośbę o skonfigurowanie projektu. **Jako nazwę wpisz `MatchGame`** (jeśli chcesz, możesz też zmienić lokalizację, w której projekt zostanie utworzony):



Kliknij przycisk *Utwórz*. Visual Studio utworzy nowy projekt o nazwie *MatchGame*.

Visual Studio utworzyło dla Ciebie katalog pełen plików

W momencie tworzenia nowego projektu Visual Studio dodało nowy katalog `MatchGame` i zapełniło go plikami oraz katalogami potrzebnymi w tym projekcie. Tu będziesz modyfikować dwa z tych plików: *MainWindow.xaml* i *MainWindow.xaml.cs*.

Ten plik zawiera kod w języku XAML definiujący interfejs użytkownika w głównym oknie.

MainWindow.xaml

Tu znajduje się kod w C# odpowiadający za działanie gry.

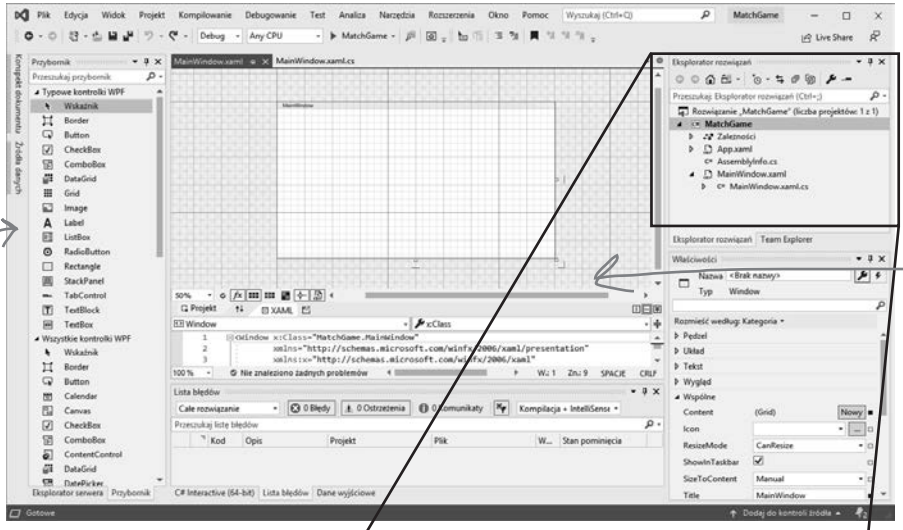
MainWindow.xaml.cs

Jeśli natrafisz na problemy z tym projektem, otwórz naszą stronę w serwisie GitHub i poszukaj odsyłacza do filmu z instruktażem: https://github.com/head-first-csharp/fourth-edition.

Zaostrz ołówek

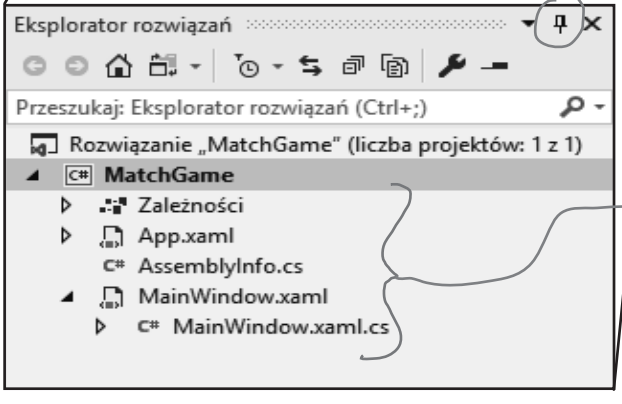
Prezentowane w książce ćwiczenia z użyciem kartki i ołówka nie są opcjonalne. Stanowią ważny aspekt nauki, praktyki i podnoszenia umiejętności z zakresu języka C#.

Dostosuj swoje środowisko IDE do zrzutu poniżej. Najpierw otwórz **plik *MainWindow.xaml***, klikając go dwukrotnie w oknie *Eksplorator Rozwiązań*. Następnie otwórz okna *Przybornik i Lista błędów*, **wybierając je w menu *Widok***. Przeznaczenie wielu okien i plików możesz określić na podstawie ich nazw i intuicji! Poświęć chwilę na **uzupełnienie pustych miejsc**. Postaraj się opisać, do czego służą poszczególne elementy IDE Visual Studio. Pierwsze pole wypełniliśmy już za Ciebie. Sprawdź, czy potrafisz odgadnąć funkcje innych komponentów.



Okno Projektant umożliwia modyfikowanie interfejsu użytkownika przez przeciąganie kontrolek.

Czy zauważyłeś, że okno Przybornik zniknęło? Kliknij ikonę pinezki, aby pozostało otwarte.

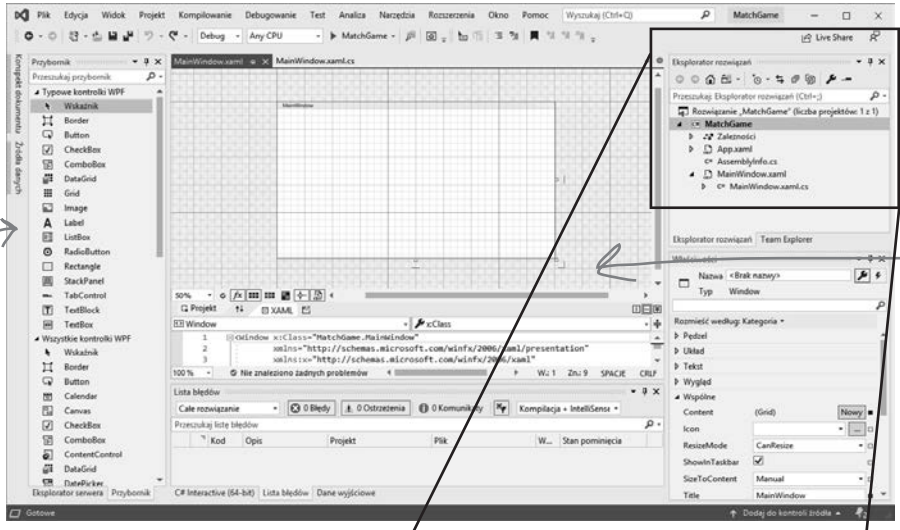


Używamy tu motywu kolorów „Jasny", aby zrzuty były bardziej czytelne. Aby zmienić motyw, wybierz opcję *Narzędzia/Opcje/Środowisko*.

Zaostrz ołówek
Rozwiązanie

Opisałeś działanie różnych komponentów IDE Visual Studio dla języka C#. Możliwe, że napisałeś coś innego niż my, ale mamy nadzieję, iż udało Ci się domyślić podstawowych funkcji każdego okna i komponentu w IDE. Nie martw się, jeśli Twoje odpowiedzi nieco różnią się od naszych! Będziesz mieć MNÓSTWO okazji do praktyki w używaniu IDE.

*Krótkie przypomnienie: w tej książce (także na tej stronie) określenia „Visual Studio" i „IDE" są stosowane **wymiennie**.*



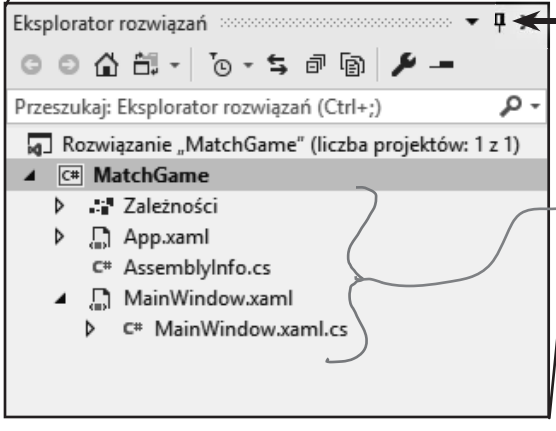
To jest Przybornik. Zawiera wiele wizualnych kontrolek, które możesz przeciągnąć do okna.

Okno Projektant umożliwia modyfikowanie interfejsu użytkownika przez przeciąganie kontrolek.

Okno Właściwości wyświetla właściwości elementu aktualnie zaznaczonego w oknie Projektant.

Okno Lista błędów informuje o błędach w kodzie. Znajdziesz tu informacje diagnostyczne na temat aplikacji.

Kliknięcie ikony pinezki powoduje włączenie lub wyłączenie automatycznego ukrywania okna. Okno Przybornik domyślnie jest automatycznie ukrywane.



Pliki C# i XAML wygenerowane przez IDE w trakcie tworzenia nowego projektu pojawiają się w Eksploratorze rozwiązań. Znajdziesz tu też inne pliki rozwiązania.

Za pomocą Eksploratora rozwiązań z IDE możesz przełączać się między plikami.

Nie istnieją głupie pytania

Zwróć uwagę na sekcje pytań i odpowiedzi. Często pomoże Ci ona rozwiać nurtujące Cię wątpliwości. Znajdziesz tu też kwestie interesujące inne osoby. Liczne z tych pytań zostały rzeczywiście zadane przez czytelników wcześniejszych wydań książki!

P: Skoro Visual Studio generuje za mnie cały kod, to czy nauka języka C# sprowadza się do nauki obsługi środowiska Visual Studio?

O: Nie. IDE świetnie sobie radzi z generowaniem fragmentów kodu, ale ma swoje ograniczenia. Niektóre zadania wykonuje bardzo dobrze — na przykład przygotowuje solidny punkt wyjścia dla programisty i automatycznie zmienia właściwości kontrolek w interfejsie użytkownika. Nie potrafi jednak najważniejszego — ustalić, co program ma robić, i sprawić, by to robił. Choć Visual Studio jest jednym z najbardziej zaawansowanych środowisk programistycznych, ma ograniczone możliwości. To ***Ty*** (nie IDE) musisz napisać kod, który wykonuje najważniejsze zadania.

P: Co zrobić, jeśli IDE wygeneruje kod, którego nie potrzebuję w projekcie?

O: Możesz zmodyfikować lub usunąć ten kod. IDE generuje kod zgodnie z tym, w jaki sposób przeciągnięte lub dodane elementy są zwykle używane. Jednak czasem potrzebujesz czegoś innego. Wszystko, co IDE zrobi dla Ciebie (każdy wiersz kodu, każdy dodany plik), można zmodyfikować — albo ręcznie, edytując pliki, albo za pomocą wygodnego interfejsu z IDE.

P: Dlaczego miałem zainstalować wersję Visual Studio Community? Na pewno nie będę potrzebować jednej z płatnych wersji, aby wykonać wszystkie zadania z tej książki?

O: W tej książce nie ma niczego, czego nie da się wykonać za pomocą bezpłatnej wersji Visual Studio (którą pobrałeś z witryny Microsoftu). Różnice między wersją Community i innymi edycjami nie przeszkadzają w pisaniu kodu w C# i tworzeniu w pełni funkcjonalnych, kompletnych aplikacji.

P: Pisaliście o łączeniu języków C# i XAML. Czym jest XAML i jak łączyć go z C#?

O: XAML jest **językiem znaczników** używanym do budowania interfejsów aplikacji WPF. XAML bazuje na XML-u, dlatego jeśli kiedykolwiek używałeś HTML-a, nauka będzie łatwiejsza. Oto przykładowy **znacznik** XAML generujący szarą elipsę:

```
<Ellipse Fill="Gray"
  Height="100" Width="75"/>
```

Jeśli w kodzie XAML projektu wpiszesz ten znacznik zaraz po znaczniku `<Grid>`, pośrodku okna pojawi się szara elipsa. Znaczniki rozpoznasz po znaku <, po którym następuje słowo (tu `Ellipse`). Jest to **początek znacznika**. Ten konkretny znacznik `Ellipse` ma trzy **właściwości**: jedna ustawia kolor elipsy na szary, a dwie pozostałe określają wysokość i szerokość figury. Kończy się on sekwencją />, ale niektóre znaczniki XAML mogą obejmować inne znaczniki. Możesz przekształcić go w **znacznik kontenerowy**, zastępując sekwencję /> znakiem >, dodając inne znaczniki (które mogą zawierać dodatkowe znaczniki) i zamykając go **znacznikiem końcowym**, który wygląda tak: `</Ellipse>`.

W trakcie lektury dowiesz się więcej o działaniu XAML-a i poznasz liczne znaczniki z tego języka.

P: Mój ekran wygląda inaczej niż na zrzutach! U mnie brakuje niektórych okien, a inne znajdują się w odmiennych miejscach. Czy zrobiłem coś nie tak? Jak mogę przywrócić ustawienia?

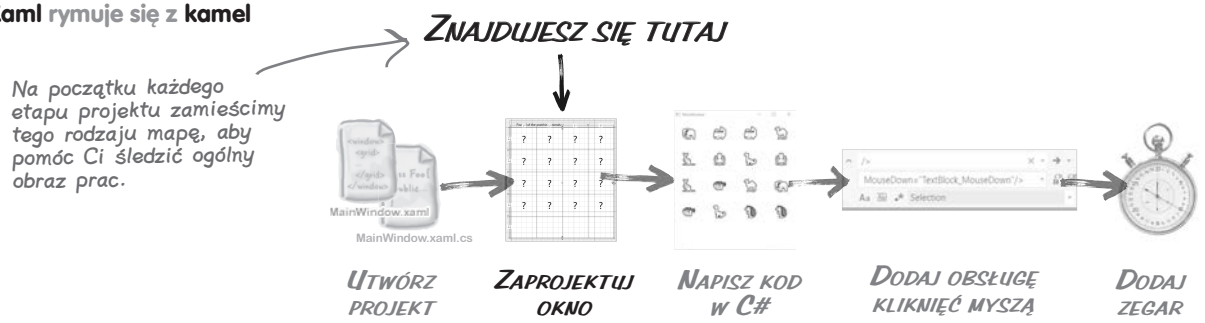
O: Jeśli wybierzesz opcję ***Resetuj układ okna*** z menu *Okno*, IDE przywróci domyślny układ okna. Następnie użyj opcji ***Widok/Inne okna***, aby otworzyć okna *Przybornik* i *Lista błędów*.

Okno Przybornik domyślnie jest zwijane. Za pomocą ikony pinezki w prawym górnym rogu Przybornika możesz spowodować, że okno to pozostanie otwarte.

Visual Studio generuje kod, który możesz wykorzystać jako punkt wyjścia do tworzenia aplikacji.

Sprawienie, by aplikacja robiła to, co ma robić, należy całkowicie do Ciebie.

Xaml rymuje się z kamel



Użyj języka XAML do zaprojektowania okna

Teraz, gdy Visual Studio utworzyło projekt WPF, pora zacząć pracę z językiem **XAML**.

XAML (od ang. **Extensible Application Markup Language**) to dający duże możliwości język znaczników, używany przez programistów języka C# do projektowania interfejsów użytkownika. Będziesz tworzyć aplikacje z użyciem dwóch rodzajów kodu. Przede wszystkim będziesz projektować interfejs użytkownika za pomocą języka XAML. Następnie dodasz kod w języku C# odpowiedzialny za działanie gry.

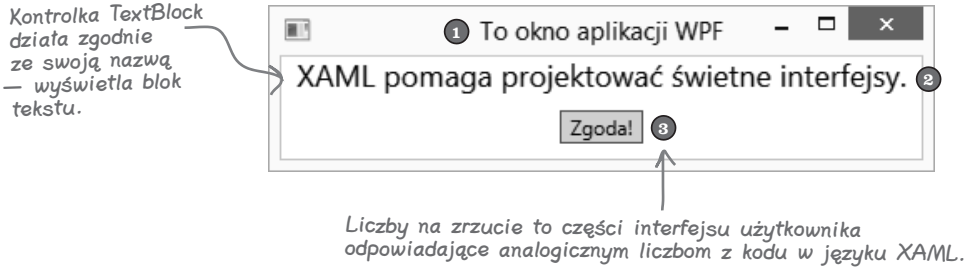
Jeśli używałeś kiedyś języka HTML do projektowania stron internetowych, dostrzeżesz dużo podobieństw w języku XAML. Oto krótki przykład ilustrujący układ okna zdefiniowany w języku XAML:

```
<Window x:Class="MyWPFApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="To okno aplikacji WPF" Height="100" Width="400"> (1)
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
        <TextBlock FontSize="18px" Text="XAML pomaga projektować świetne interfejsy."/> (2)
        <Button Width="50" Margin="5,10" Content="Zgoda!"/> (3)
    </StackPanel>
</Window>
```

Dodaliśmy liczby do fragmentów kodu definiujących tekst.

Poszukaj analogicznych liczb na zrzucie poniżej.

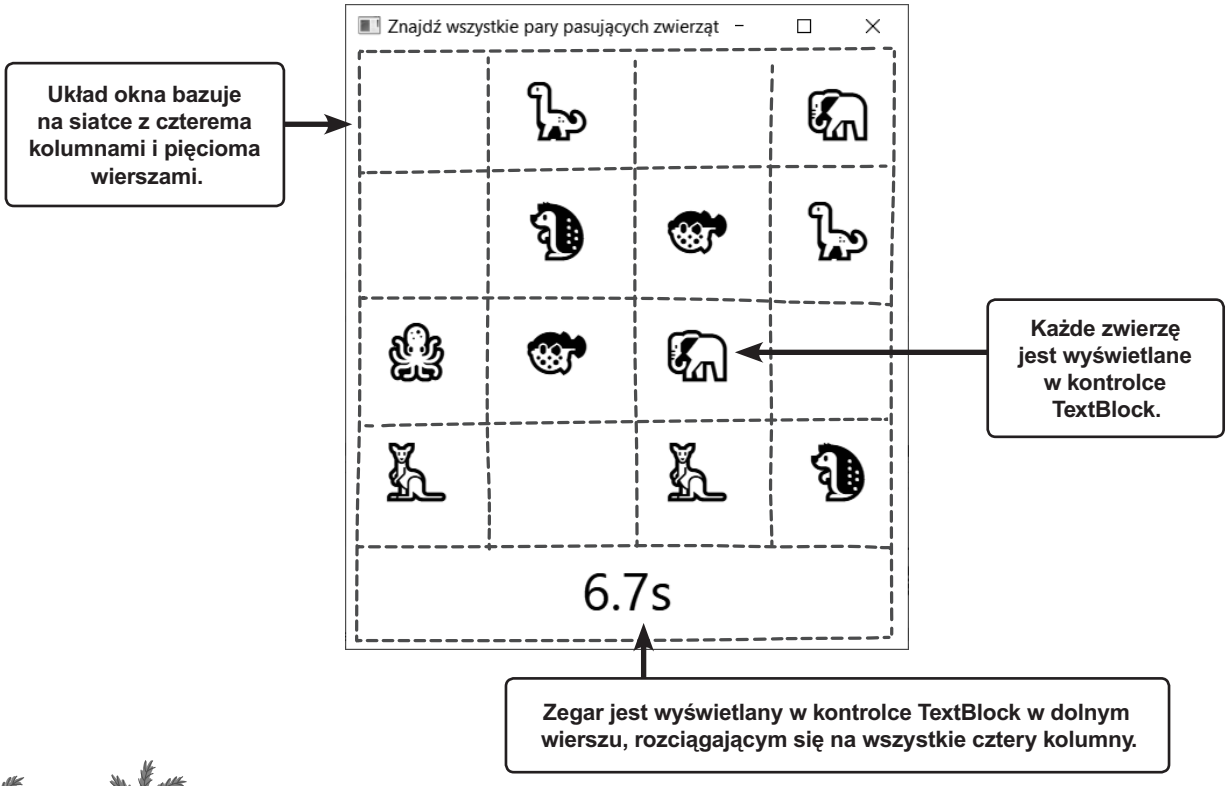
Poniżej pokazano, jak platforma WPF **renderuje** (wyświetla na ekranie) to okno. Platforma generuje okno z dwoma widocznymi **kontrolkami**: kontrolką `TextBlock`, która wyświetla tekst, i kontrolką `Button`, którą użytkownik może kliknąć. Te elementy są umieszczone w niewidocznej kontrolce `StackPanel`, która powoduje wyświetlanie kontrolek jedna pod drugą. Przyjrzyj się kontrolkom na zrzucie, a następnie wróć do kodu w języku XAML i znajdź w nim znaczniki `TextBlock` i `Button`.



Zaprojektuj okno gry

Będziesz potrzebować aplikacji z graficznym interfejsem użytkownika, obiektów odpowiedzialnych za działanie gry i pliku wykonywalnego do uruchamiania. Wydaje się, że wymaga to dużo pracy, ale w pozostałej części rozdziału przygotujesz wszystkie elementy. W efekcie tego opanujesz na niezłym poziomie projektowanie atrakcyjnie wyglądających aplikacji WPF.

Oto układ okna aplikacji, jaką utworzysz:



Znajomość języka XAML jest ważna dla programistów języka C#.

Możesz sobie myśleć tak: „Chwileczkę! To książka *C#. Rusz głową*. Po co przeznaczać tyle czasu na XAML? Nie lepiej skupić się na C#?”.

W aplikacjach WPF (a także innych projektach w języku C#) do projektowania interfejsu użytkownika używany jest język XAML. XAML stosuje się nie tylko w aplikacjach desktopowych; w aplikacjach mobilnych dla systemów Android i iOS używana jest platforma Xamarin Forms, która używa odmiany języka XAML (z *odrobinę* innym zestawem kontrolek). To dlatego umiejętność tworzenia interfejsów użytkownika w języku XAML jest ważna dla każdego programisty języka C# i dlatego w tej książce dowiesz się o wiele więcej na temat XAML-a. Zobaczysz ***krok po kroku***, jak pisać kod XAML. Możesz też użyć XAML-owego *Projektanta* z Visual Studio 2019, aby przygotować interfejs użytkownika bez pisania dużej ilości kodu. ***A zatem, aby wszystko było jasne***:

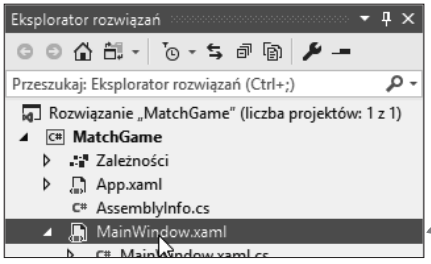
XAML to język do definiowania interfejsu użytkownika. C# służy do definiowania działania programu.

Ustaw wielkość okna i podaj nagłówek za pomocą właściwości w XAML-u

Zacznij tworzenie interfejsu użytkownika gry w dopasowywanie zwierząt. W pierwszej kolejności zmniejsz szerokość i nagłówek okna. Zaznajomisz się też z XAML-owym *Projektantem* — rozbudowanym narzędziem do projektowania atrakcyjnych interfejsów użytkownika swoich aplikacji.

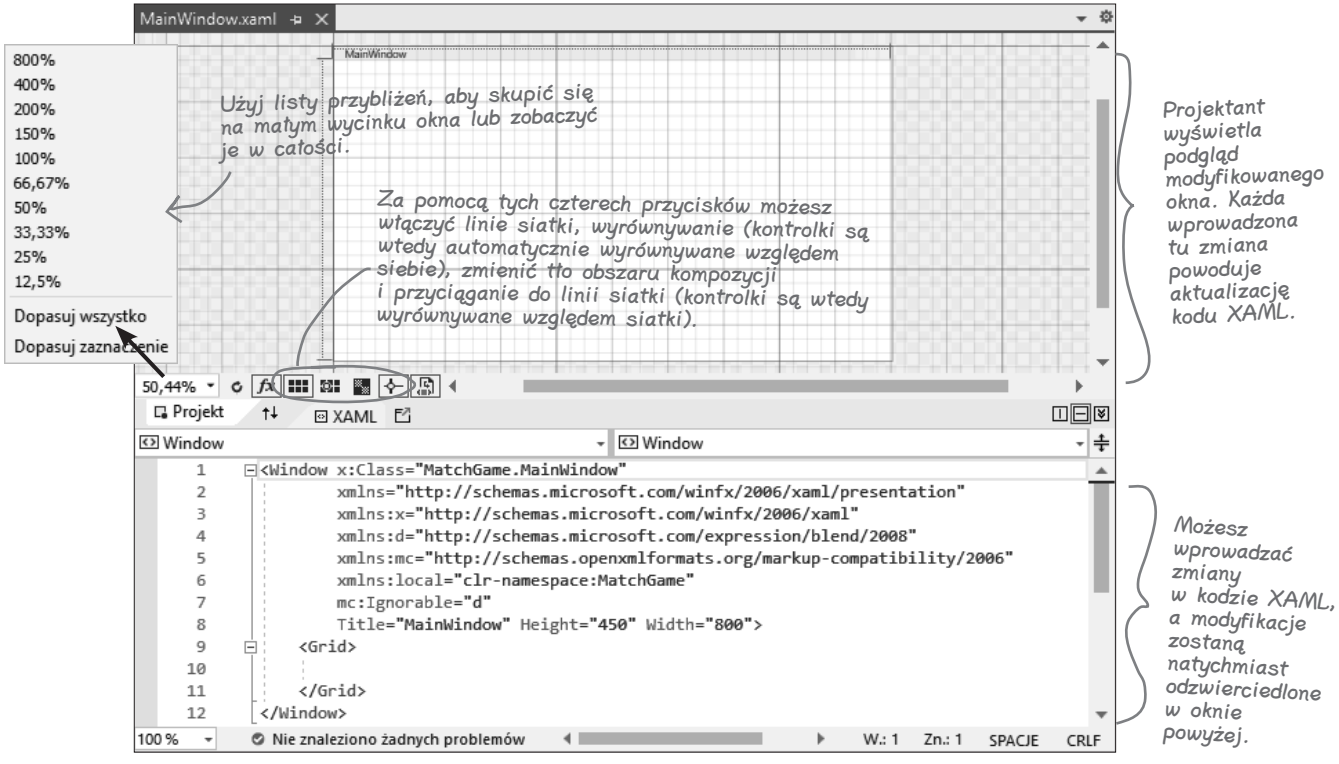
1 **Wybierz plik *MainWindow*.**

Kliknij dwukrotnie plik *MainWindow.xaml* w *Eksploratorze rozwiązań*.



Kliknij dwukrotnie plik w Eksploratorze rozwiązań, aby otworzyć go w odpowiednim edytorze. Pliki z kodem C# (rozszerzenie .cs) są otwierane w edytorze kodu. Pliki XAML (rozszerzenie .xaml) są otwierane w XAML-owym projektancie.

Gdy tylko to zrobisz, Visual Studio otworzy plik w XAML-owym *Projektancie*.



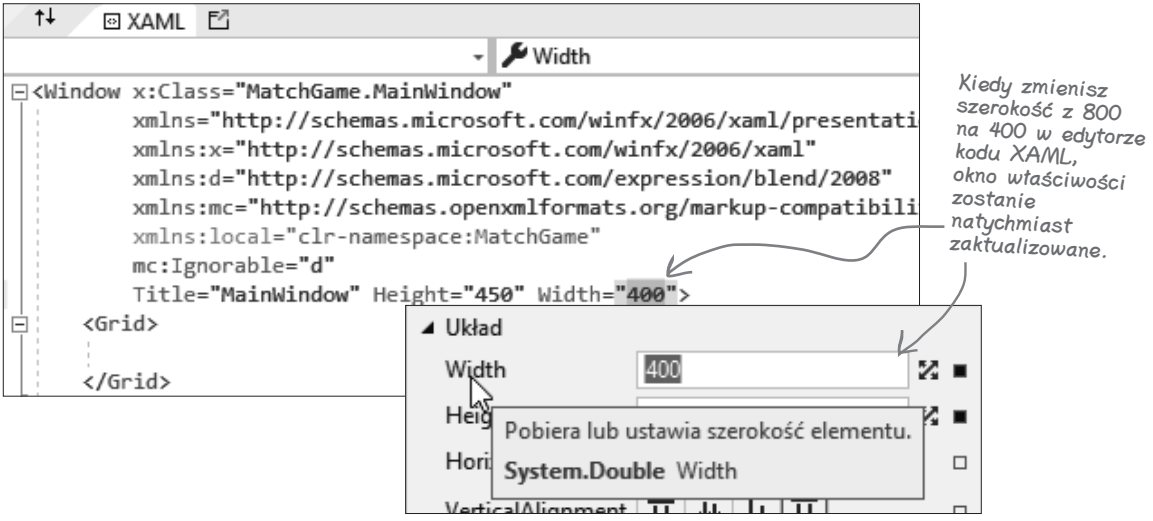
Projektant wyświetla podgląd modyfikowanego okna. Każda wprowadzona tu zmiana powoduje aktualizację kodu XAML.

Możesz wprowadzać zmiany w kodzie XAML, a modyfikacje zostaną natychmiast odzwierciedlone w oknie powyżej.

2 **Zmień wielkość okna.**

Przenieś kursor do edytora kodu XAML i kliknij dowolny z pierwszych ośmiu wierszy kodu. W oknie z właściwościami powinieneś zobaczyć wtedy właściwości okna.

Rozwiń sekcję *Układ* i **zmień szerokość na 400**. Okno na panelu *Projektanta* natychmiast stanie się węższe. Spójrz na kod XAML — właściwość `Width` jest teraz równa `400`.



3 **Zmień nagłówek okna.**

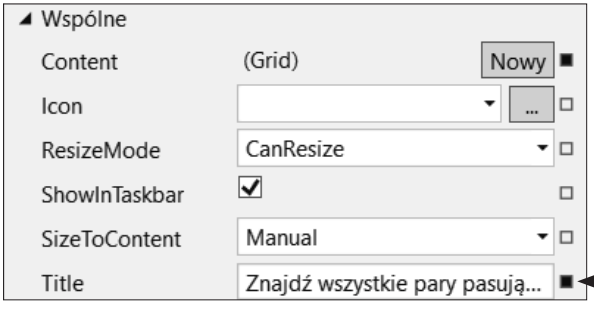
Znajdź w kodzie XAML ten wiersz (znajduje się on na końcu znacznika `Window`):

```
Title="MainWindow" Height="450" Width="400">
```

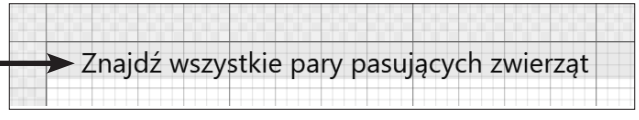
Zmień tytuł (właściwość `Title`) na **`Znajdź wszystkie pary pasujących zwierząt`**:

```
Title="Znajdź wszystkie pary pasujących zwierząt" Height="450" Width="400">
```

Zobaczysz zmianę w sekcji *Wspólne* okna *Właściwości*, jednak ważniejsze jest to, że na pasku tytułu w oknie widoczny jest nowy tekst.



Gdy modyfikujesz właściwości w znacznikach XAML, zmiany są natychmiast odzwierciedlane w oknie Właściwości. Gdy używasz okna Właściwości do zmiany interfejsu użytkownika, IDE aktualizuje kod XAML.

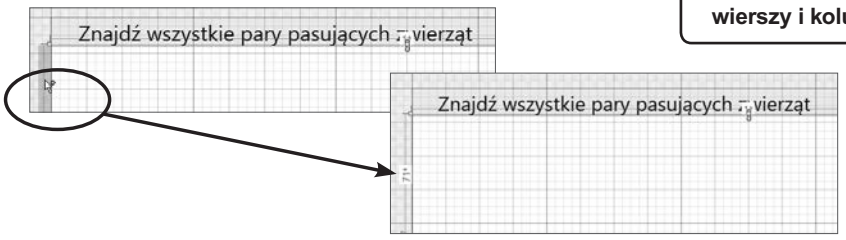


Dodaj wiersze i kolumny do siatki w XAML-u

Może się wydawać, że główne okno jest puste. Przyjrzyj się jednak dokładniej dolnej części kodu XAML. Czy zauważyłeś, że znajdują się tam wiersze **<Grid>** i **</Grid>**? W oknie znajduje się **siatka**, która jednak na razie jest niewidoczna, ponieważ nie obejmuje żadnych wierszy ani kolumn. Zaraz dodasz do niej wiersz.

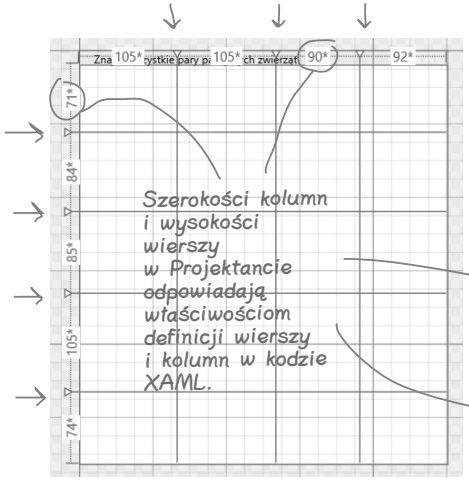
Przenieś kursor nad lewą stronę *Projektanta*. Gdy obok kursora pojawi się znak plus, kliknij myszą, aby dodać wiersz.

Interfejs użytkownika aplikacji WPF jest zbudowany z kontrolek (przycisków, etykiet, pól wyboru itd.). Siatka jest kontenerem, czyli kontrolką specjalnego rodzaju, która może zawierać inne kontrolki. Siatka pozwala użyć wierszy i kolumn do zdefiniowania układu okna.



Zobaczysz liczbę z gwiazdką i poziomą linię w oknie. Właśnie dodałeś wiersz do siatki! Teraz dodaj wiersze i kolumny:

- ★ Wykonaj wcześniejszą operację cztery razy, aby uzyskać w sumie pięć wierszy.
- ★ Umieść kursor w górnej części okna i kliknij, aby dodać cztery kolumny. Okno powinno wyglądać tak jak na zrzucie poniżej (widoczne liczby będą zapewne inne — nie stanowi to problemu).
- ★ Wróć do kodu XAML. Znajdują się w nim teraz znaczniki `ColumnDefinition` i `RowDefinition` odpowiadające dodanym wierszom i kolumnom.




```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="105*"/>
    <ColumnDefinition Width="105*"/>
    <ColumnDefinition Width="90*"/>
    <ColumnDefinition Width="92*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="71*"/>
    <RowDefinition Height="84*"/>
    <RowDefinition Height="85*"/>
    <RowDefinition Height="105*"/>
    <RowDefinition Height="74*"/>
</Grid.RowDefinitions>
```

Ramki „Obejrzyj to!" zwracają uwagę na ważne, ale często niejasne kwestie, które mogą utrudnić lub spowolnić pracę.

Obejrzyj to!

W Twoim IDE ekran może wyglądać nieco inaczej.

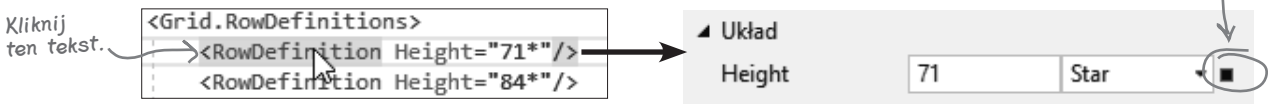
*Wszystkie zrzuty z tej książki zostały wykonane w środowisku **Visual Studio Community 2019 dla systemu Windows**. Jeśli używasz wersji Professional lub Enterprise, możesz dostrzec drobne różnice.*

Nie przejmuj się — niezależnie od wersji, program będzie działał tak samo.

Wyrównaj wielkości wierszy i kolumn

Gdy gra wyświetla graczowi zwierzęta do dopasowania, rysunki powinny być równo rozmieszczone. Każde zwierzę będzie znajdować się w komórce siatki, a sama siatka będzie automatycznie dostosowywana do wielkości okna. Dlatego wiersze i kolumny powinny być równej wielkości. Na szczęście XAML umożliwia łatwą zmianę wielkości wierszy i kolumn. **Kliknij pierwszy znacznik `RowDefinition` w edytorze kodu XAML**, aby wyświetlić jego ustawienia w oknie *Właściwości*:

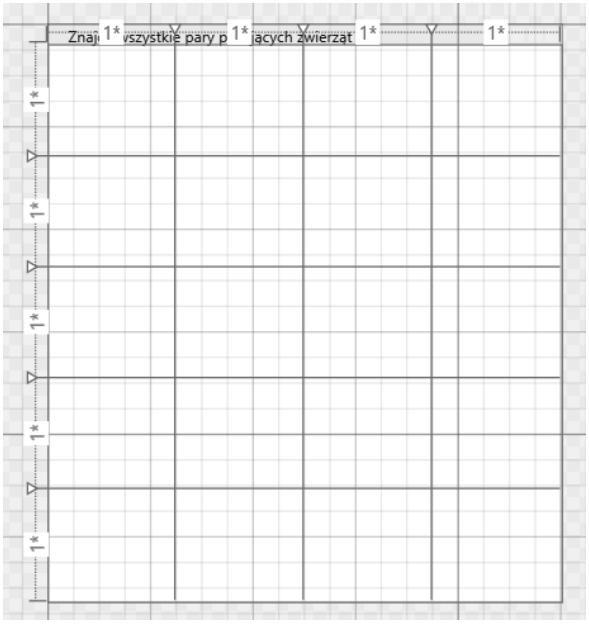
Gdy ten kwadracik jest wypełniony, wartość właściwości jest inna od domyślnej. Aby przywrócić ustawienia domyślne, kliknij kwadracik i wybierz w menu opcję Resetuj.



Przejdź do okna *Właściwości* i **kliknij kwadracik** na prawo od właściwości *Height*, po czym **wybierz opcję *Resetuj* z menu**, które się pojawi. Chwileczkę! Powoduje to zniknięcie wiersza w *Projektancie*! No cóż, tak naprawdę wiersz nie zniknął, stał się tylko bardzo niski. **Zresetuj właściwość *Height*** wszystkich wierszy. Następnie **zresetuj właściwość *Width*** wszystkich kolumn. Siatka powinna teraz mieć cztery kolumny i pięć wierszy o równej wielkości.

Naprawdę postaraj się przeczytać kod XAML. Jeśli nie pracowałeś wcześniej w językach HTML lub XML, początkowo kod może wydawać się serią niezrozumiałych <nawiasów> i /ukośników. Jednak im więcej będziesz mieć z nim do czynienia, tym bardziej będzie dla Ciebie zrozumiały.

Oto co powinieneś zobaczyć w Projektancie:



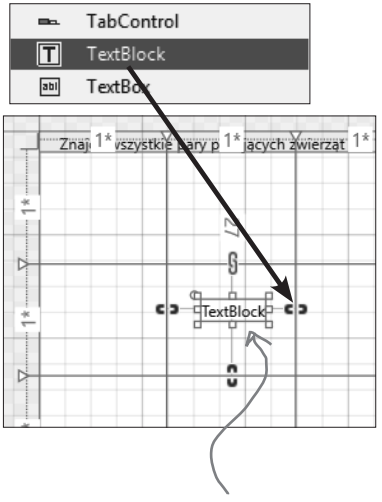
A to powinieneś zobaczyć w edytorze kodu XAML między otwierającym znacznikiem `<Window...>` i zamykającym znacznikiem `</Window>`:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
</Grid>
```

To kod XAML tworzący siatkę z czterema równymi kolumnami i pięcioma równymi wierszami.

Dodaj kontrolki TextBlock do siatki

WPF używa **kontrolek `TextBlock`** do wyświetlania tekstu. My użyjemy ich do wyświetlania dopasowywanych zwierząt. Dodaj taką kontrolkę do okna.



Gdy przeciągniesz kontrolkę TextBlock z Przybornika do komórki, IDE doda tę kontrolkę do kodu XAML i ustawi dla niej wiersz, kolumnę oraz margines.

Rozwiń sekcję *Typowe kontrolki WPF* w *Przyborniku* i **przeciągnij kontrolkę `TextBlock` do komórki w drugiej kolumnie drugiego wiersza**. IDE doda znacznik `TextBlock` między początkowym i końcowym znacznikiem `Grid`:

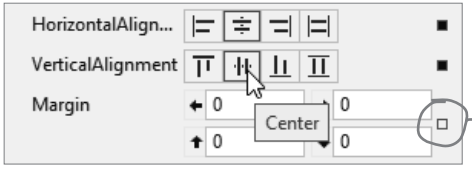
```
<TextBlock Text="TextBlock"
    HorizontalAlignment="Left" VerticalAlignment="Center"
    Margin="560,0,0,0" TextWrapping="Wrap" />
```

W kodzie XAML tej kontrolki `TextBlock` jest pięć właściwości:

- ★ `Text` określa tekst wyświetlany w oknie.
- ★ `HorizontalAlignment` wyrównuje tekst do lewej, do prawej lub względem środka.
- ★ `VerticalAlignment` wyrównuje tekst względem góry, środka lub dołu.
- ★ `Margin` ustawia margines względem krawędzi kontenera.
- ★ `TextWrapping` określa, czy tekst w kontrolce ma być zawijany.

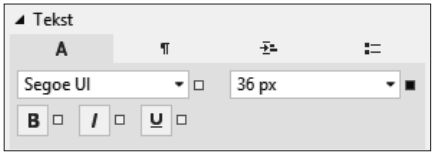
U Ciebie właściwości mogą być podane w innej kolejności, a właściwość *Margin* może mieć inną wartość, ponieważ zależy ona od miejsca w komórce, w które przeciągnąłeś kontrolkę. Wszystkie te właściwości można zmieniać i resetować w oknie *Właściwości* w IDE.

Każde zwierzę ma być wyśrodkowane. **Kliknij etykietę** w *Projektancie*, a następnie w oknie *Właściwości* kliknij Układ, aby rozwinąć **sekcję *Układ***. Kliknij ***Center*** dla obu właściwości, *HorizontalAlignment* i *VerticalAlignment*, a następnie użyj kwadracika po prawej stronie, aby **zresetować właściwość *Margin***.

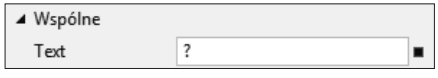


Kliknij kwadracik i wybierz opcję Resetuj, aby zresetować marginesy.

Ponadto zwierzęta mają być większe, dlatego **rozwiń sekcję *Tekst*** w oknie *Właściwości* i **zmień wielkość czcionki** na `36 px`. Następnie otwórz sekcję *Wspólne* i zmień właściwość *Text* na `?`, aby wyświetlany był znak zapytania.



Właściwość Text (w sekcji Wspólne) określa tekst kontrolki TextBlock.



Kliknij pole wyszukiwania w górnej części okna *Właściwości* i wpisz słowo `wrap`, aby znaleźć pasujące właściwości. Użyj kwadracika po prawej stronie, aby zresetować właściwość *TextWrapping*.

Nie istnieją głupie pytania

P: Po zresetowaniu wysokości pierwszych czterech wierszy wszystkie one znikają. Jednak kiedy zresetuję wysokość ostatniego, wszystkie wiersze znów są widoczne. Dlaczego tak się dzieje?

O: Wiersze wyglądały tak, jakby zniknęły, ponieważ w siatce WPF domyślnie używane jest **proporcjonalne ustawianie wielkości** wierszy i kolumn. Jeśli wysokość ostatniego wiersza wynosi 74*, to po ustawieniu domyślnej wysokości pierwszych czterech wierszy na 1* siatka zmienia ich wielkość na 1/78 (1,3%) wysokości siatki. Ostatni wiersz zajmuje wtedy 74/78 (94,8%) wysokości, dlatego pierwsze cztery wiersze wyglądają na bardzo niskie. Gdy ustawisz dla ostatniego wiersza wysokość domyślną 1*, siatka zmieni wielkość każdego wiersza na 20% wysokości siatki.

P: Gdy ustawiam szerokość okna na 400, co oznacza ta wartość? Jak szerokie jest to 400?

O: WPF używa **pikseli niezależnych od urządzenia**, zawsze równych 1/96 cala. To oznacza, że 96 pikseli na *nieprzeskalowanym* wyświetlaczu zawsze jest równe 1 cal. Jeśli jednak zmierzysz okno linijką, może się okazać, że nie zajmuje dokładnie 400 pikseli (4,16 cala). To dlatego, że Windows używa przydatnego mechanizmu zmieniającego skalę elementów na ekranie. Dzięki temu aplikacje nie są malutkie nawet wtedy, gdy wyświetlasz je, używając telewizora po drugiej stronie pokoju jako monitora komputera. Piksele niezależne od urządzenia pomagają platformie WPF zapewnić dobry wygląd aplikacji w każdej skali.

W książce natrafisz na wiele ćwiczeń tego rodzaju. Umożliwiają Ci one rozwój umiejętności programistycznych. Zawsze dopuszczalne jest podejrzenie rozwiązania!

Ćwiczenie

Masz już jedną kontrolkę `TextBlock` — to dobry punkt wyjścia! Jednak potrzebnych jest 16 takich kontrolek, aby wyświetlić wszystkie dopasowywane zwierzęta. Czy wiesz, jak dodać kod XAML, aby umieścić identyczne kontrolki XAML w komórkach z pierwszych czterech wierszy siatki?

Zacznij od przyjrzenia się znacznikowi XAML, jaki utworzyłeś. Powinien wyglądać podobnie do poniższego. Właściwości mogą być podane w innej kolejności, a tu rozdzieliliśmy je na dwa wiersze (też możesz to zrobić, jeśli chcesz, aby kod XAML był czytelniejszy):

```
<TextBlock Text="?" Grid.Column="1" Grid.Row="1" FontSize="36"
    HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

Twoje zadanie polega na **skopiowaniu tej kontrolki `TextBlock`**, aby każda z pierwszych 16 komórek siatki zawierała identyczną kontrolkę. Aby ukończyć ćwiczenie, musisz więc *dodać do aplikacji jeszcze 15 kontrolek TextBlock*. Pamiętaj o kilku kwestiach:

- Wiersze i kolumny są numerowane od 0, co jest też wartością domyślną. Dlatego jeśli pominiesz właściwości `Grid.Row` lub `Grid.Column`, kontrolka `TextBlock` pojawi się w pierwszym od lewej wierszu lub w górnej kolumnie.
- Interfejs użytkownika możesz edytować w *Projektancie*. Możesz też skopiować i wkleić kod XAML. Wypróbuj obie te techniki i zobacz, która jest dla Ciebie najlepsza.

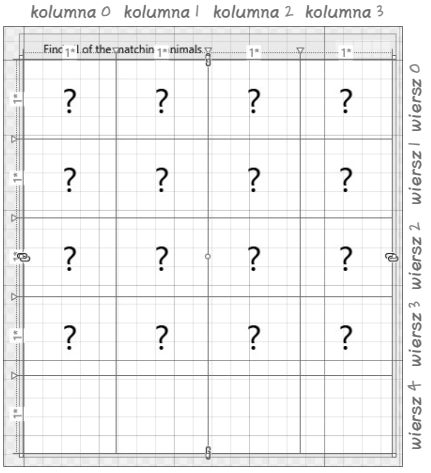
Ćwiczenie

Oto kod XAML 16 kontrolek TextBlock na dopasowywane zwierzęta. Wszystkie te kontrolki są prawie identyczne. Różnią się tylko właściwościami `Grid.Row` i `Grid.Column`, które pozwalają umieścić po jednej kontrolce `TextBlock` w każdej z 16 pierwszych komórek siatki o wymiarach 5 na 4.
Znacznik `Window` się nie zmienił, dlatego został pominięty w tym miejscu.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>

  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
    <RowDefinition/>
  <RowDefinition/>
</Grid.RowDefinitions>
```

Tak wygląda definicje kolumn i wierszy po zrównaniu wielkości tych elementów.

Tak wygląda okno w Projektancie w Visual Studio po dodaniu wszystkich kontrolek TextBlock.




```
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="1"/>
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="2"/>
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="3"/>

<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Row="1"/>
<TextBlock Text="?" FontSize="36" Grid.Row="1" Grid.Column="1"
    HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="1" Grid.Column="2"
    HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="1" Grid.Column="3"
    HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

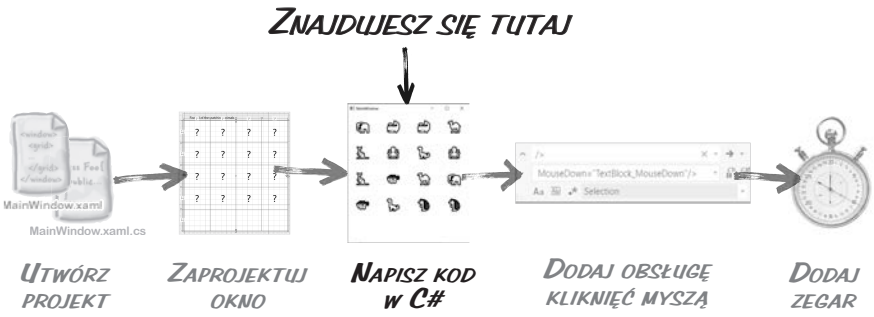
W tych czterech kontrolkach TextBlock właściwość Grid.Row ma wartość 1, ponieważ znajdują się w drugim wierszu od góry (pierwszy wiersz ma numer 0).

```
<TextBlock Text="?" FontSize="36" Grid.Row="2" HorizontalAlignment="Center" VerticalAlignment="Center"/>

<TextBlock Text="?" FontSize="36" Grid.Row="2" Grid.Column="1"
    HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="2" Grid.Column="2"
    HorizontalAlignment="Center" VerticalAlignment="Center"/>
<TextBlock Text="?" FontSize="36" Grid.Row="2" Grid.Column="3"
    HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

Dopuszczalne jest dodanie właściwości Grid.Row lub Grid.Column o wartości 0. Tu pominęliśmy takie właściwości, ponieważ 0 jest wartością domyślną.

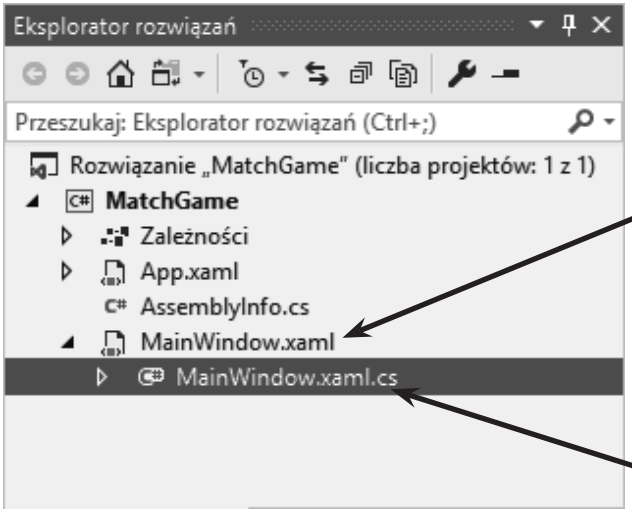
```
  <TextBlock Text="?" FontSize="36" Grid.Row="3" HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="3" Grid.Column="1"
      HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="3" Grid.Column="2"
      HorizontalAlignment="Center" VerticalAlignment="Center"/>
  <TextBlock Text="?" FontSize="36" Grid.Row="3" Grid.Column="3"
      HorizontalAlignment="Center" VerticalAlignment="Center"/>
</Grid>
```

Kodu jest dużo, jednak jest to powtórzony 16 razy ten sam wiersz z drobnymi zmianami. Każdy wiersz zaczynający się od `<TextBlock` ma te same cztery właściwości (`Text`, `FontSize`, `HorizontalAlignment` i `VerticalAlignment`). Wiersze różnią się tylko właściwościami `Grid.Row` i `Grid.Column`. Właściwości mogą być podawane w dowolnej kolejności.



Teraz możesz zacząć pisać kod gry

Zakończyłeś projektowanie głównego okna, a przynajmniej wystarczającej jego części, aby przejść do następnego aspektu działania gry. Pora dodać kod C# odpowiedzialny za przebieg rozgrywki.



Edytowałeś kod XAML pliku *MainWindow.xaml*. To tam znajdują się wszystkie elementy projektowe okna. Kod XAML z tego pliku definiuje wygląd i układ okna.

Teraz zaczniesz pracę nad kodem C#. Znajdzie się on w pliku *MainWindow.xaml.cs*. Jest to kod zaplecza (ang. *code behind*) dla okna, ponieważ działa razem ze znacznikami z pliku XAML. To dlatego nazwy plików są prawie identyczne (różnią się tylko dodatkowym członem *.cs* na końcu). Do tego pliku dodasz kod C# definiujący działanie gry, w tym kod umieszczający emoji w siatce, obsługujący kliknięcia myszą i kontrolujący zegar.

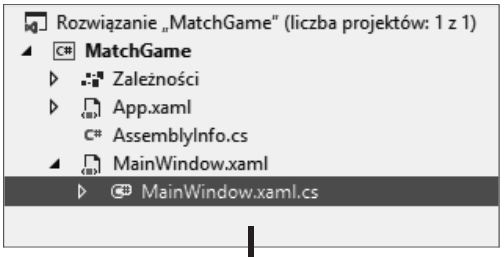
Obejrzyj to!

Gdy wpisujesz kod C#, musi on być w pełni poprawny.

*Część osób uważa, że programistą tak naprawdę stajesz się dopiero wtedy, gdy pierwszy raz godzinami szukasz błędnie umieszczonej kropki. Wielkość liter też jest ważna: `SetUpGame` to nie to samo co `setUpGame`. Nadmiarowe przecinki, średniki, nawiasy itd. mogą sprawić, że kod przestanie działać, lub — co jeszcze gorsze — wpłynąć na kod tak, że wprawdzie się skompiluje, ale będzie funkcjonować niezgodnie z oczekiwaniami. **Wykorzystujący sztuczną inteligencję mechanizm IntelliSense** z IDE pomaga unikać takich problemów, ale nie wykona za Ciebie całej pracy.*

Wygeneruj metodę konfigurującą grę

Wygeneruj to!



Za pomocą zakładek w górnej części okna możesz przełączać się między edytorem kodu C# a Projektantem kodu XAML.

Teraz, po przygotowaniu interfejsu użytkownika, pora zacząć pisać kod gry. W tym celu **wygenerujesz metodę** (taką jak metoda `Main`, którą poznałeś wcześniej), a następnie dodasz do niej kod.

1 **Otwórz plik MainWindow.xaml.cs w edytorze.**

Kliknij trójkąt (▷) obok nazwy *MainWindow.xaml* w *Eksploratorze rozwiązań*, a następnie **kliknij dwukrotnie plik *MainWindow.xaml.cs***, aby otworzyć go w edytorze kodu w IDE. Zauważ, że plik zawiera już kod. Visual Studio pomoże Ci dodać do niego metodę.

To nie problem, jeśli na razie nie masz pewności, czym jest metoda.

2 **Wygeneruj metodę o nazwie SetUpGame.**

Znajdź następujący fragment w otwartym kodzie:

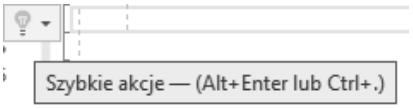
```
public MainWindow();
{
    InitializeComponent();
}
```

Kliknij na końcu wiersza `InitializeComponent();`, aby kursor znalazł się zaraz po średniku. Wciśnij dwukrotnie *Enter*, a następnie wpisz: `SetUpGame();`

Bezpośrednio po wpisaniu średnika nazwa `SetUpGame` zostanie podkreślona falowaną linią. Kliknij tekst `SetUpGame`. W oknie powinna pojawić się ikona żółtej żarówki. Kliknij ją, aby **otworzyć menu szybkich akcji**, i użyj go do wygenerowania metody.

Okno Podgląd zmian informuje o błędzie, który spowodował pojawienie się czerwonej falowanej linii, a także pokazuje podgląd kodu, jaki akcja wygeneruje w celu wyeliminowania usterki.

Kiedy klikniesz ikonę szybkich akcji, zobaczysz menu kontekstowe z możliwymi działaniami. Jeśli dana operacja generuje kod, zobaczysz jego podgląd. Wybierz akcję Generuj metodę, aby wygenerować nową metodę `SetUpGame`.



Ikona żółtej żarówki informuje o tym, że zaznaczyłeś kod, dla którego dostępna jest szybka akcja. Oznacza to, że istnieje zadanie, które Visual Studio może zautomatyzować. Możesz kliknąć żółtą żarówkę lub wcisnąć kombinację *Alt+Enter* albo *Ctrl+.* (kropka), aby zobaczyć dostępne akcje.

3 Spróbuj uruchomić kod.

Kliknij przycisk w górnej części środowiska IDE, aby uruchomić program — tak jak wcześniej zrobiłeś to z aplikacją konsolową.

Przycisk „Rozpocznij debugowanie" na pasku narzędzi w górnej części IDE uruchamia aplikację. Możesz też wybrać opcję Rozpocznij debugowanie (F5) w menu Debugowanie.

Hmm — coś poszło nie tak. Zamiast wyświetlić okno, **program zgłosił wyjątek**:

```
21      public partial class MainWindow : Window
22      {
            Odwołania: 0
23          public MainWindow()
24          {
25              InitializeComponent();
26              SetUpGame();
27          }
28
            1 odwołanie
29          private void SetUpGame()
30          {
31              throw new NotImplementedException();
32          }
33      }
34  }
```

Wyjątek obsłużony przez użytkownika

System.NotImplementedException: „The method or operation is not implemented."

Wyświetl szczegóły | Kopiuj szczegóły | Rozpocznij sesję rozszerzenia Live Share

Ustawienia wyjątków

Może się wydawać, że program jest uszkodzony, ale w rzeczywistości można było oczekiwać takiego efektu! IDE wstrzymało program i wyróżniło ostatni uruchomiony wiersz kodu. Przyjrzyj się następującej instrukcji:

```
throw new NotImplementedException();
```

Metoda wygenerowana przez IDE dosłownie nakazuje językowi C# zgłosić wyjątek. Spójrz na komunikat powiązany z wyjątkiem:

```
System.NotImplementedException: "The method or operation is not implemented."
```

Ma to sens, ponieważ **to Ty musisz zaimplementować metodę** wygenerowaną przez IDE. Jeśli zapomniałeś to zrobić, wyjątek jest uprzejmym przypomnieniem, że masz jeszcze kod do napisania. Jeżeli generujesz dużo metod, dobrze jest mieć takie przypomnienie!

Kliknij kwadrat *Zatrzymaj debugowanie* () na pasku narzędzi (lub wybierz opcję *Zatrzymaj debugowanie* w menu *Debugowanie*), aby zatrzymać program, co pozwoli Ci zakończyć implementowanie metody `SetUpGame`.

Gdy używasz IDE do uruchamiania aplikacji, przycisk Zatrzymaj debugowanie powoduje natychmiastowe jej zamknięcie.

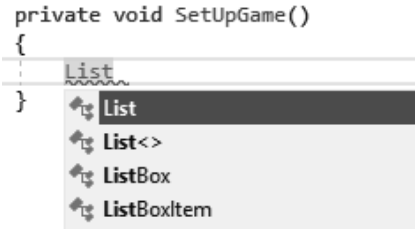
Dokończ metodę SetUpGame

Jest to specjalna metoda nazywana konstruktorem. Więcej o działaniu takich metod dowiesz się z rozdziału 5.

Metoda `SetUpGame` znajduje się w metodzie `public MainWindow()`, ponieważ cały kod z metody `MainWindow` jest wykonywany bezpośrednio po uruchomieniu aplikacji.

1 **Zacznij dodawać kod do metody SetUpGame.**

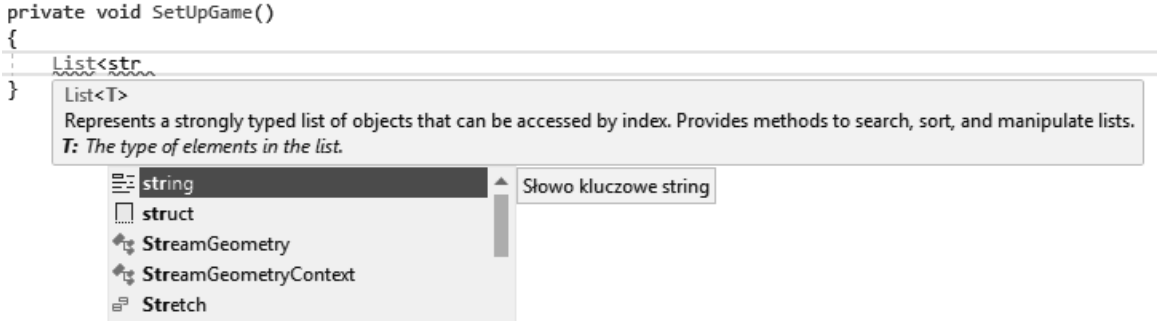
Metoda `SetUpGame` ma losowo przypisywać osiem par emoji przedstawiających zwierzęta do kontrolek `TextBlock`, aby gracz mógł dopasowywać rysunki. Dlatego metoda potrzebuje listy emoji, a w napisaniu kodu pomoże Ci IDE. Zaznacz instrukcję `throw` dodaną przez IDE i usuń ją. Następnie umieść kursor w miejscu, gdzie znajdowała się ta instrukcja, i wpisz `List`. IDE wyświetli **okno mechanizmu IntelliSense** z zestawem słów kluczowych rozpoczynających się członem `List`:



Wkrótce dowiesz się dużo więcej o metodach.

Właśnie użyłeś IDE, aby pomogło Ci dodać metodę do aplikacji. Nie przejmuj się, jeśli nadal nie masz pewności, czym jest metoda. Znacznie więcej o metodach i strukturze kodu w języku C# dowiesz się z następnego rozdziału.

Wybierz pozycję `List` z listy IntelliSense. Następnie wpisz `<str`. Pojawi się nowe okno IntelliSense z pasującymi słowami kluczowymi:



Wybierz `string`. Dokończ wpisywanie poniższego wiersza kodu, ale **jeszcze nie wciskaj *Enter***:

```
List<string> animalEmoji = new List<string>()
```

Lista jest kolekcją, która przechowuje zestaw wartości w określonym porządku. Więcej o kolekcjach dowiesz się z rozdziałów 8. i 9.

Słowo kluczowe „new" służy do tworzenia kolekcji List. Więcej o tym słowie dowiesz się w rozdziale 3.

② Dodawanie wartości do kolekcji List.

Instrukcja w C# nie jest jeszcze gotowa. Upewnij się, że kursor znajduje się zaraz po nawiasie) kończącym wiersz, a następnie wpisz otwierający nawias klamrowy {. IDE doda nawias zamykający za Ciebie i umieści kursor pomiędzy nawiasami. **Wciśnij Enter**. IDE automatycznie doda znaki podziału wiersza.

```
List<string> animalEmoji = new List<string>()
{

}
```

Gdy otwarty jest panel emoji, możesz wpisać słowo takie jak „ośmiornica", a zostanie ono zastąpione odpowiednim emoji.

Użyj **panelu emoji systemu Windows** (klawisz systemu *Windows+kropka*) lub przejdź do ulubionej witryny z emoji (na przykład *https://emojipedia.org/nature*) i skopiuj jedno emoji. Wróć do kodu, dodaj symbol ", następnie wklej emoji, wpisz kolejny symbol ", przecinek, spację i następny symbol ", jeszcze raz wklej to samo emoji, po czym wprowadź ostatni symbol " i przecinek. Następnie zrób to samo dla siedmiu kolejnych emoji. Celem jest uzyskanie **w nawiasie klamrowym ośmiu par emoji ze zwierzętami**. Dodaj **;** po zamykającym nawiasie klamrowym:

```
List<string> animalEmoji = new List<string>()
{
    "🐵","🐵",
    "🐶","🐶",
    "🐱","🐱",
    "🦁","🦁",
    "🐯","🐯",
    "🐴","🐴",
    "🦄","🦄",
    "🐮","🐮",
};
```

Umieść kursor nad kropkami pod nazwą animalEmoji. IDE poinformuje, że przypisana wartość nigdy nie jest używana. Ostrzeżenie zniknie zaraz po tym, jak użyjesz listy emoji w dalszym kodzie metody.

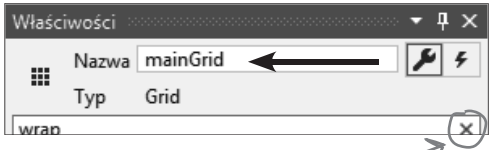
Panel emoji jest wbudowany w system Windows 10. Aby wyświetlić ten panel, wystarczy wcisnąć jednocześnie klawisz Windows i kropkę.

③ Ukończ metodę.

Teraz dodaj do metody **pozostały kod**. *Uważaj* na kropki, nawiasy zwykłe i klamrowe.

```
Random random = new Random();

foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
{
    int index = random.Next(animalEmoji.Count);
    string nextEmoji = animalEmoji[index];
    textBlock.Text = nextEmoji;
    animalEmoji.RemoveAt(index);
}
```

Ten wiersz powinien znajdować się zaraz po zamykającym nawiasie klamrowym i średniku.



Nie zapomnij wyzerować pola wyszukiwania.

Czerwona falowana linia pod nazwą `mainGrid` w IDE informuje o błędzie. Program nie skompiluje się, ponieważ w kodzie nie ma niczego o tej nazwie. **Wróć do edytora XAML** i kliknij znacznik `<Grid>`, następnie przejdź do okna *Właściwości* i wpisz `mainGrid` w polu *Name*.

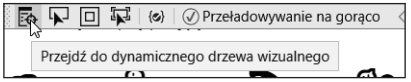
Sprawdź kod XAML. Na początku kodu siatki zobaczysz znacznik `<Grid x:Name="mainGrid">`. Teraz w kodzie nie powinno być żadnych błędów. Jeśli są, ***starannie sprawdź każdy wiersz*** — łatwo jest coś przeoczyć.

Jeśli po uruchomieniu gry wystąpi wyjątek, upewnij się, że na liście animalEmoji znajduje się dokładnie osiem par emoji, a w kodzie XAML jest 16 znaczników <TextBlock .../>.

Uruchom program

Kliknij przycisk ▶ Start na pasku narzędzi IDE, aby uruchomić program. Pojawi się okno z ośmioma parami zwierząt w losowych lokalizacjach:

Gdy pierwszy raz uruchomisz program, zobaczysz narzędzia środowiska uruchomieniowego w górnej części okna:

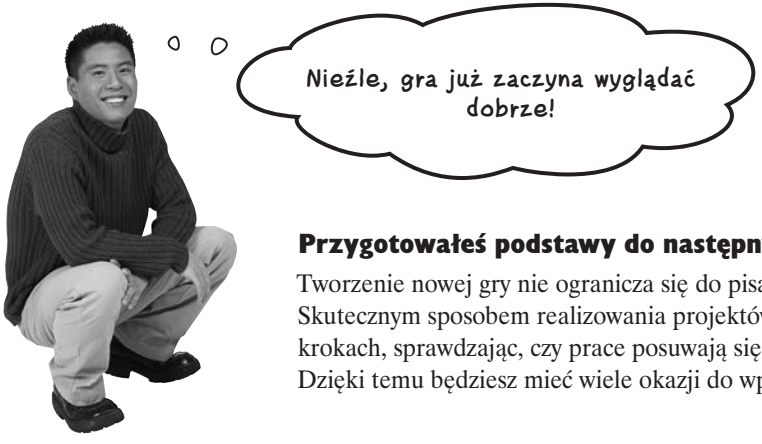


Kliknij pierwszy przycisk z narzędzi, aby wyświetlić panel *Dynamiczne drzewo wizualne* w IDE:

Następnie kliknij pierwszy przycisk panelu *Dynamiczne drzewo wizualne*, aby wyłączyć narzędzia środowiska uruchomieniowego.

Gdy program działa, IDE przechodzi w tryb debugowania. Przycisk *Start* jest zastępowany nieaktywnym (jasnoszarym) przyciskiem *Kontynuuj*, a na pasku narzędzi pojawiają się **kontrolki debugowania** (⏸ ■ ↻) z przyciskami do wstrzymywania, zatrzymywania i ponownego uruchamiania programu.

Zakończ program, klikając znak *X* w prawym górnym rogu okna lub kwadracik *Zatrzymaj* w kontrolkach debugowania. Uruchom program kilkakrotnie. Zwierzęta za każdym razem powinny pojawiać się gdzie indziej.



Przygotowałeś podstawy do następnych dodawanych elementów.

Tworzenie nowej gry nie ogranicza się do pisania kodu. Realizujesz też projekt. Skutecznym sposobem realizowania projektów jest tworzenie ich w małych krokach, sprawdzając, czy prace posuwają się w dobrym kierunku. Dzięki temu będziesz mieć wiele okazji do wprowadzenia poprawek.

To następne ćwiczenie z ołówkiem i kartką. Zdecydowanie warto poświęcić czas na wykonanie wszystkich takich ćwiczeń, ponieważ pomogą Ci one szybciej opanować ważne zagadnienia związane z językiem C#.

KTO CO ROBI?

Gratulacje — napisałeś działający program! To prawda, programowanie to coś dużo więcej niż kopiowanie kodu z książki. Jednak nawet jeśli nigdy wcześniej nie pisałeś kodu, może zaskoczyć Cię, jak duże jego fragmenty już rozumiesz. Narysuj linię łączącą każdą z instrukcji w C# (po lewej) z opisami ich działania (po prawej). Pierwszy punkt wykonaliśmy już za Ciebie.

Instrukcje C#

```
List<string> animalEmoji = new List<string>()
{
    "🐵","🐵",
    "🐶","🐶",
    "🐘","🐘",
    "🐮","🐮",
    "🐫","🐫",
    "🦒","🦒",
    "🐰","🐰",
    "🐉","🐉",
};
```

```
Random random = new Random();
```

```
foreach (TextBlock textBlock in
    mainGrid.Children.OfType<TextBlock>())
```

```
int index = random.Next(animalEmoji.Count);
```

```
string nextEmoji = animalEmoji[index];
```

```
textBlock.Text = nextEmoji;
```

```
animalEmoji.RemoveAt(index);
```

Działanie kodu

Przypisuje do kontrolki `TextBlock` losowe emoji z listy.

Znajduje wszystkie kontrolki `TextBlock` z głównej siatki i wykonuje dla każdej z nich podane instrukcje.

Usuwa losowe emoji z listy.

Tworzy listę ośmiu par emoji.

Wybiera liczbę losową z przedziału od 0 do liczby emoji pozostałych na liście i nazywa tę wartość „index”.

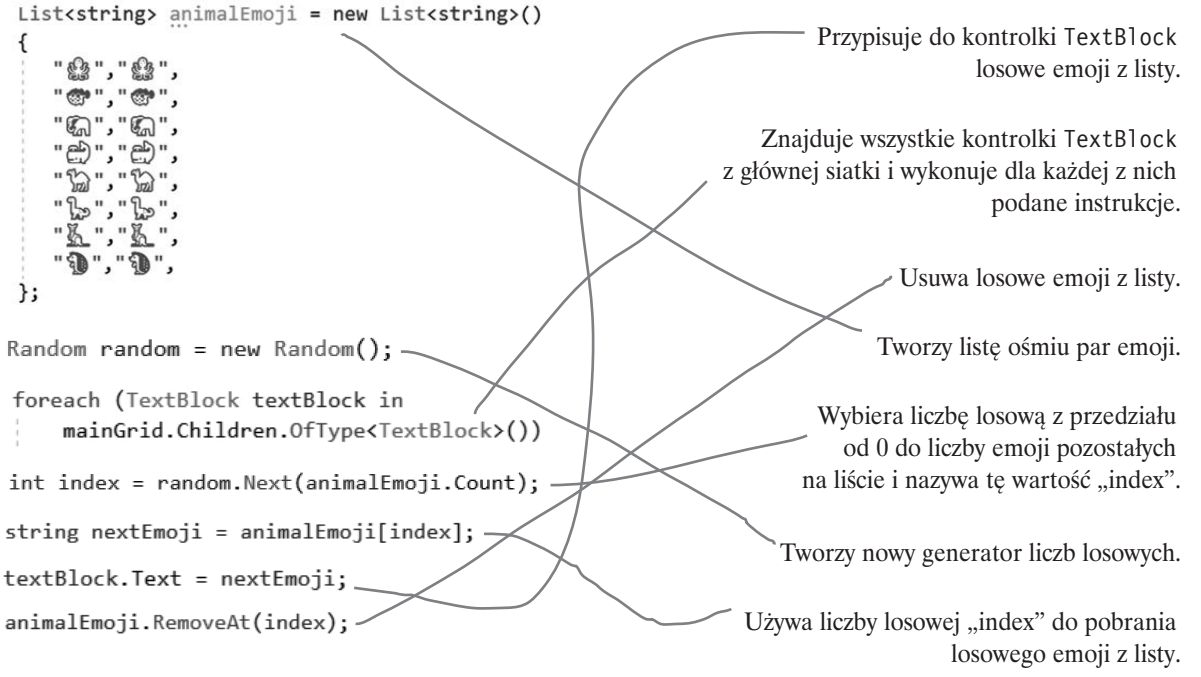
Tworzy nowy generator liczb losowych.

Używa liczby losowej „index” do pobrania losowego emoji z listy.

KTO CO ROBI?

ROZWIĄZANIE

Instrukcje C# | **Działanie kodu**



MINI

Zaostrz ołówek

Oto ćwiczenie z ołówkiem i kartką, które pomoże Ci dobrze zrozumieć napisany kod w C#.

1. Weź kartkę i ułóż ją w orientacji poziomej. Narysuj pośrodku pionową linię.
2. Po lewej stronie zapisz całą metodę `SetUpGame`. Między poszczególnymi instrukcjami pozostaw puste miejsce. Nie musisz dbać o precyzję przy rysowaniu emoji.
3. Po prawej stronie kartki obok odpowiednich instrukcji zapisz każdą odpowiedź z ćwiczenia „kto co robi?”. Przeczytaj tekst po obu stronach linii — kod powinien nabrać sensu.

Nie jestem przekonana co do zadań „Zaostrz ołówek" i ćwiczenia z dopasowywaniem. Czy nie byłoby lepiej, gdybyście po prostu **pokazali mi kod** do wpisania w IDE?

Rozwijanie umiejętności zrozumienia kodu sprawi, że staniesz się lepszym programistą.

Ćwiczenie z użyciem ołówka i kartki **nie są opcjonalne**. Zapewniają Twojemu mózgowi inny sposób przyswajania informacji. Robią też coś jeszcze ważniejszego — dają Ci możliwość ***popełniania błędów***. Pomyłki są częścią procesu uczenia się i każdy z nas robił wiele bęłdów (możliwe nawet, że w tej książce znajdziesz jakąś literówkę lub dwie!). Nikt nie pisze od razu doskonałego kodu. Naprawdę dobrzy programiści zawsze zakładają, że kod, który piszą dziś, jutro może wymagać zmian. Dalej w książce zapoznasz się z procesem *refaktoryzacji*. Jest to technika programowania, która polega na ulepszaniu kodu już po jego napisaniu.

Dodajemy tego rodzaju listy wypunktowane, aby zapewnić krótkie streszczenie wielu opisanych do danego miejsca zagadnień i narzędzi.

CELNE SPOSTRZEŻENIA

- Visual Studio jest **IDE Microsoftu** (**zintegrowanym środowiskiem programistycznym**), które upraszcza i wspomaga edycję plików z kodem C# i zarządzanie nimi.
- **Aplikacje konsolowe .NET Core** to działające w różnych systemach aplikacje z tekstowymi danymi wejściowymi i wyjściowymi.
- **Wykorzystujący sztuczną inteligencję mechanizm IntelliSense** w IDE pomaga w szybszym wprowadzeniu kodu.
- **WPF** (ang. *Windows Presentation Foundation*) to technologia tworzenia w C# aplikacji z interfejsem graficznym.
- Interfejsy użytkownika w technologii WPF są projektowane w języku **XAML** (ang. *eXtensible Application Markup Language*). Jest to bazujący na XML-u język znaczników, w którym znaczniki i właściwości służą do definiowania kontrolek interfejsu użytkownika.
- **Kontrolka XAML `Grid`** służy do tworzenia układu siatki do przechowywania innych kontrolek.
- **Znacznik XAML `TextBlock`** reprezentuje kontrolkę do przechowywania tekstu.
- **Okno *Właściwości*** w IDE umożliwia łatwe modyfikowanie właściwości kontrolek — na przykład układu, tekstu albo wiersza lub kolumny siatki, gdzie kontrolka się znajduje.

Dodaj nowy projekt do systemu kontroli wersji

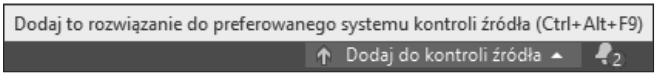
W tej książce będziesz tworzyć wiele różnych projektów. Czy nie byłoby świetnie, gdyby istniał łatwy sposób na archiwizowanie ich i łatwy dostęp do nich z dowolnego miejsca? Co zrobisz, jeśli popełnisz błąd — czy nie byłoby wygodnie, gdybyś mógł odzyskać starszą wersję kodu? No cóż, masz szczęście! Tak właśnie działają **systemy kontroli wersji** — umożliwiają łatwe zarchiwizowanie całego kodu i śledzenie wszystkich wprowadzonych zmian. Visual Studio pozwala na łatwe dodawanie projektów do takich systemów.

Git to popularny system kontroli wersji, a Visual Studio może zapisywać kod źródłowy w dowolnym **repozytorium** Git. Uważamy, że **GitHub** jest jednym z najłatwiejszych w użyciu dostawców systemu Git. Aby umieszczać kod w serwisie GitHub, musisz założyć w nim konto. Jeśli jeszcze nie masz takiego konta, otwórz stronę *https://github.com* i utwórz je.

Dodawanie projektu do systemu kontroli wersji jest opcjonalne.

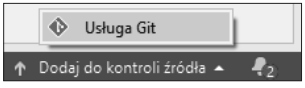
Możliwe, że pracujesz na komputerze w sieci firmowej, która nie zapewnia dostępu do serwisu GitHub (jest to polecany przez nas dostawca systemu Git). Możliwe też, że nie masz ochoty używać takiego systemu. Niezależnie od powodów możesz pominąć ten krok. Możesz też umieścić kod w prywatnym repozytorium, jeśli chcesz utworzyć kopię zapasową, ale nie chcesz, by inne osoby mogły znaleźć kod.

Znajdź opcję ***Dodaj do kontroli źródła*** na pasku stanu w dolnej części IDE:

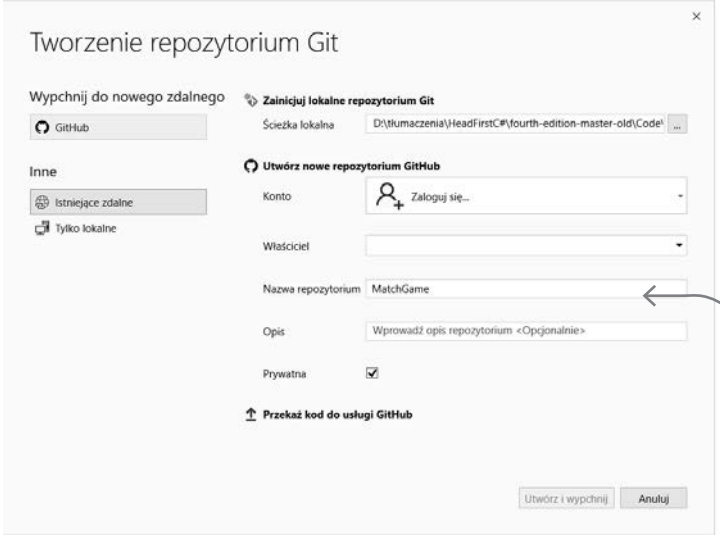


Zaraz po dodaniu kodu do systemu Git pasek stanu zmieni się, informując, że kod projektu znajduje się w systemie kontroli wersji. Git jest bardzo popularnym systemem kontroli wersji, a Visual Studio obejmuje kompletnego klienta tego systemu. Katalog projektu zawiera teraz ukryty katalog *.git* używany do śledzenia wszystkich poprawek w kodzie.

Kliknij tę opcję. Visual Studio pozwoli dodać kod do systemu Git:



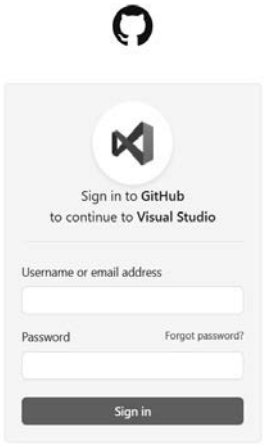
Kliknij *Usługa Git*. Visual Studio wyświetli okno ***Tworzenie repozytorium Git***.



Visual Studio utworzy repozytorium w Twoim koncie w serwisie GitHub. Domyślnie nazwa repozytorium jest taka sama jak nazwa projektu.

Git jest otwartym systemem kontroli wersji. Istnieje wiele niezależnych serwisów takich jak GitHub, które udostępniają usługi związane z systemem Git (na przykład przestrzeń dyskową na kod i dostęp do repozytorium z poziomu internetu). Więcej o systemie Git dowiesz się na stronie *https://git-scm.com*.

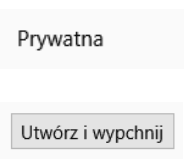
Kliknij Zaloguj się... . Visual Studio otworzy w przeglądarce **formularz logowania w serwisie GitHub**. Wprowadź nazwę użytkownika i hasło do serwisie GitHub. Jeśli skonfigurowałeś uwierzytelnianie dwuetapowe, zostaniesz poproszony o jego użycie. Po zalogowaniu się może pojawić się prośba o przyznanie serwisowi GitHub do środowiska Visual Studio. Jeśli zobaczysz taką prośbę, przyznaj uprawnienia, aby umożliwić środowisku Visual Studio tworzenie repozytoriów i przesyłanie kodu.



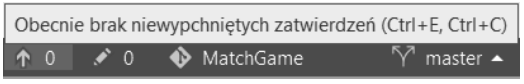
Upewnij się, że wpisana jest Twoja nazwa użytkownika z serwisu GitHub.

Po zalogowaniu się do serwisu GitHub status systemu Git na pasku stanu zmieni się. Zobaczysz, że nie ma już **zatwierdzeń** (zapisanych wersji kodu), które nie zostały ***wypchnięte*** do lokalizacji poza Twoim komputerem. To oznacza, że projekt jest zsynchronizowany z repozytorium z Twojego konta w serwisie GitHub.

Kliknij **przycisk *Utwórz i wypchnij***, aby utworzyć nowe repozytorium w serwisie GitHub i opublikować w nim kod. Po wypchnięciu kodu do serwisu GitHub

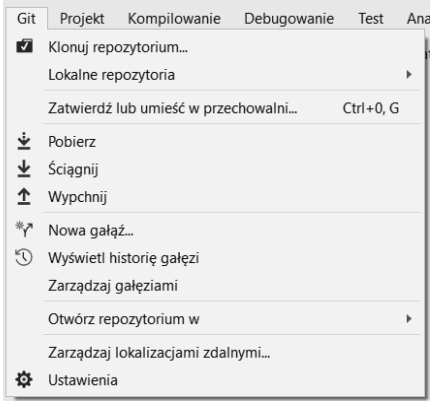


Po zalogowaniu się do serwisu GitHub użyj tego przycisku do opublikowania projektu na swoim koncie.



Zwróć uwagę na pasek stanu. Jeśli zobaczysz liczbę, na przykład ↑ 2, jest to informacja, że masz dwa niewypchnięte zatwierdzenia, które możesz przesłać do repozytorium w serwisie GitHub.

Po opublikowaniu kodu w serwisie GitHub możesz używać poleceń z menu Git do pracy z repozytorium systemu Git



Otwórz stronę *https://github.com/<twoja-nazwa-w-github>/MatchGame*, aby zobaczyć przesłany kod. Gdy zsynchronizujesz projekt ze zdalnym repozytorium, zobaczysz aktualizacje w sekcji z zatwierdzeniami.

Nie istnieją głupie pytania

P: Czy kod XAML to prawdziwy kod?

O: Tak, jak najbardziej. Pamiętasz czerwone falowane linie pod nazwą `mainGrid` w kodzie C#, które zniknęły dopiero po dodaniu nazwy do znacznika `Grid` w kodzie XAML? Stało się tak, ponieważ zmodyfikowałeś kod. Po dodaniu nazwy do kodu XAML kod C# mógł jej użyć.

P: Myślałem, że XAML działa jak HTML, który jest interpretowany przez przeglądarkę. Czy XAML działa tak samo?

O: Nie, kod XAML jest rozwijany razem z kodem C#. W następnym rozdziale zobaczysz, że można użyć słowa kluczowego `partial` do podziału klasy na kilka plików. Ten mechanizm służy też do złączania kodu XAML i C#. Kod XAML definiuje interfejs użytkownika, a C# definiuje operacje, a oba te komponenty są złączane za pomocą klas częściowych.

Dlatego ważne jest, aby traktować kod XAML jak zwykły kod. Z tego samego powodu opanowanie języka XAML jest ważne dla każdego programisty używającego C#.

P: Zauważyłem MNÓSTWO wierszy `using` na początku pliku z kodem C#. Dlaczego jest ich aż tyle?

O: Aplikacje WPF korzystają z kodu z różnych *przestrzeni nazw* (w następnym rozdziale dowiesz się, czym one są). Gdy Visual Studio tworzy projekt WPF, automatycznie dodaje na początku pliku *MainWindow.xaml.cs* **dyrektywy `using`** dla najczęściej używanych przestrzeni nazw. Z niektórych przestrzeni nazw już korzystałeś. IDE używa jaśniejszej czcionki do wyróżniania przestrzeni nazw, które nie są wykorzystywane w kodzie.

P: Aplikacje desktopowe wydają się dużo bardziej skomplikowane od konsolowych. Czy naprawdę oba typy aplikacji działają tak samo?

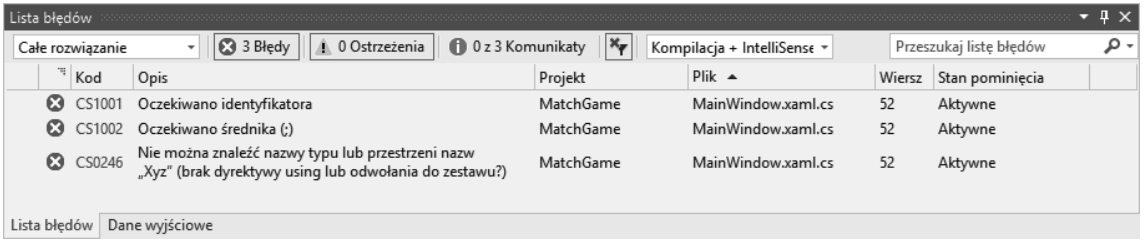
O: Tak. Na podstawowym poziomie cały kod C# działa tak samo: wykonywana jest jedna instrukcja, potem druga itd. Aplikacje desktopowe wydają się bardziej skomplikowane, ponieważ niektóre metody są wywoływane tylko wtedy, gdy dzieją się określone rzeczy — na przykład wyświetlane jest okno lub użytkownik kliknął przycisk. Gdy metoda już zostanie wywołana, działa dokładnie tak samo jak w aplikacji konsolowej.

Wskazówka na temat IDE: Lista błędów

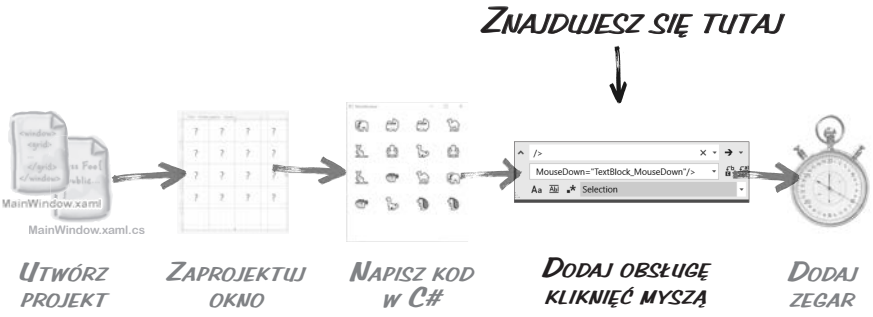
Spójrz na dolną część edytora kodu. Zwróć uwagę na komunikat Nie znaleziono żadnych problemów. Oznacza to, że kod można **skompilować**. Kompilacja polega na przekształcaniu kodu przez IDE na postać **binarną**, którą system operacyjny może uruchomić. Wprowadź teraz błąd w kodzie.

Przejdź do pierwszego wiersza nowej metody `SetUpGame`. Wciśnij dwukrotnie klawisz *Enter*, a następnie w nowym wierszu wpisz `Xyz`.

Ponownie spójrz na dolną część edytora kodu. Teraz widnieje tam komunikat 3. Jeśli okno ***Lista błędów*** nie jest widoczne, otwórz je, wybierając je w menu ***Widok***. Na liście błędów znajdują się teraz trzy błędy:

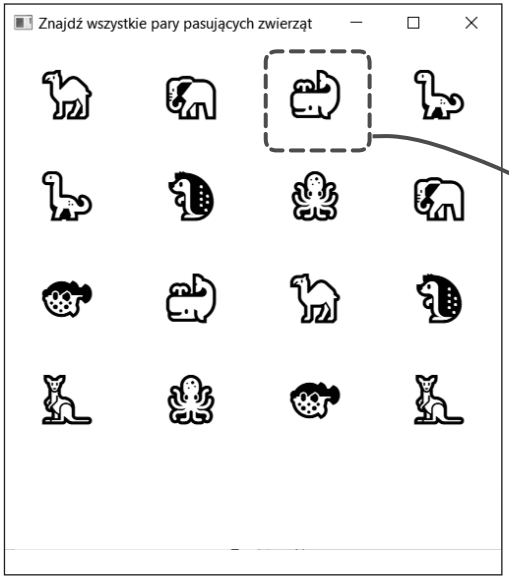


IDE wyświetliło informacje o błędach, ponieważ tekst Xyz nie jest poprawnym kodem w języku C# i uniemożliwia kompilację kodu. Dopóki w kodzie znajdują się błędy, nie będzie on działać. Dlatego usuń dodany wiersz Xyz.



Następny krok tworzenia gry to dodanie obsługi kliknięć myszą

Gdy gra wyświetla już zwierzęta, które gracz może kliknąć, trzeba dodać kod odpowiadający za przebieg gry. Gracz ma klikać pary zwierząt. Pierwsze kliknięte zwierzę znika. Jeśli drugie kliknięte zwierzę pasuje do pierwszego, także przestaje być widoczne. Jeśli jednak kliknięte rysunki nie pasują do siebie, program ma je ponownie wyświetlić. Aby umożliwić działanie tego procesu, dodasz **procedurę obsługi zdarzeń**. Taka procedura to metoda wywoływana, gdy w aplikacji zajdą określone zdarzenia (kliknięcie myszą, dwukrotne kliknięcie, zmiana wielkości okna itd.).



Gdy gracz kliknie jedno ze zwierząt, aplikacja wywoła metodę TextBlock_MouseDown, która obsługuje kliknięcia myszą. Oto, co ta metoda będzie robić.

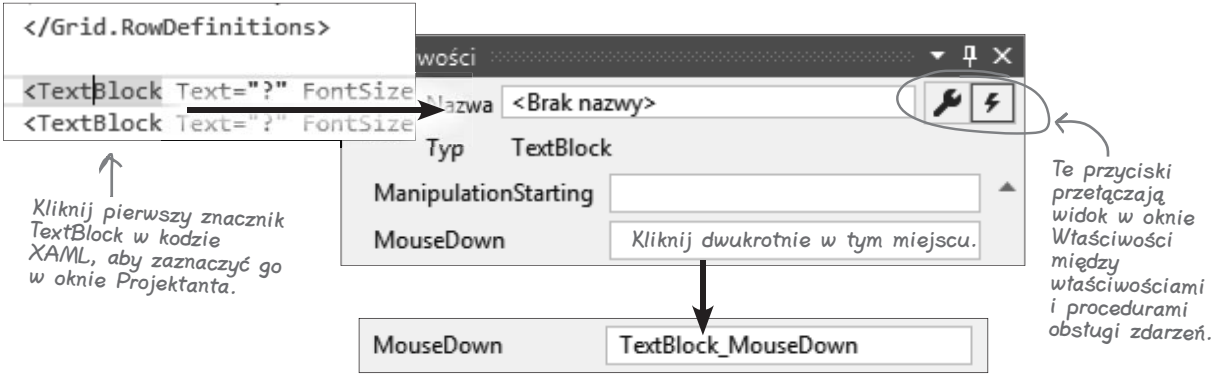
```
TextBlock_MouseDown() {

  /* Jeśli kliknięte zostało pierwsze zwierzę,
   * należy zapisać, która kontrolka TextBlock
   * została kliknięta i usunąć grafikę. Jeżeli
   * kliknięto drugie zwierzę, należy albo też
   * usunąć grafikę (dopasowanie), albo ponownie
   * wyświetlić pierwszy rysunek (brak dopasowania).
   */
}
```

To jest komentarz. Wszystko między znakami / i */ jest ignorowane przez C#. Ten komentarz został dodany, aby opisać działanie metody TextBlock_MouseDown, a także by pokazać, jak wyglądają komentarze.*

Spraw, aby kontrolki TextBlock reagowały na kliknięcia myszą

Metoda `SetUpGame` modyfikuje kontrolki `TextBlock`, aby wyświetlały emoji ze zwierzętami. Wiesz już więc, w jaki sposób za pomocą kodu modyfikować kontrolki w aplikacji. Teraz napiszesz kod innego rodzaju, który będzie wywoływany przez kontrolki. Pomoże Ci w tym IDE.

Wróć do edytora kodu XAML i **kliknij pierwszy znacznik `TextBlock`**. IDE zaznaczy powiązaną kontrolkę w *Projektancie* i będziesz mógł zmienić jej właściwości. W oknie *Właściwości* kliknij przycisk *Procedury obsługi zdarzeń* ( ). **Procedura obsługi zdarzeń** jest metodą wywoływaną przez aplikację po zajściu określonych zdarzeń. Takie zdarzenia to wciśnięcie klawisza, przeciągnięcie elementu, zmiana wielkości okna lub, naturalnie, ruchy i kliknięcia myszą. Przewiń okno *Właściwości* w dół i przyjrzyj się nazwom różnych zdarzeń kontrolki `TextBlock`, dla jakich można dodać procedury obsługi. **Kliknij dwukrotnie pole na prawo od zdarzenia `MouseDown`**.



IDE zapełni pole `MouseDown` nazwą metody, `TextBlock_MouseDown`, a w kodzie XAML danej kontrolki `TextBlock` znajdzie się właściwość `MouseDown`:

```
<TextBlock Text="?" FontSize="36" HorizontalAlignment="Center"
    VerticalAlignment="Center" MouseDown="TextBlock_MouseDown"/>
```

Może tego nie zauważyłeś, ponieważ IDE ponadto **dodało nową metodę** do kodu zaplecza (powiązanego z kodem XAML) i natychmiast otworzyło edytor kodu C#, aby wyświetlić tę metodę. Zawsze możesz wrócić do tej metody z edytora kodu XAML, klikając prawym przyciskiem myszy właściwość `TextBlock_MouseDown` i wybierając opcję *Pokaż kod*. Oto ta metoda:

```
private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{

}
```

Gdy gracz kliknie tę kontrolkę `TextBlock`, aplikacja automatycznie wywoła metodę `TextBlock_MouseDown`. Dlatego teraz wystarczy dodać kod tej metody. Następnie będziesz musiał powiązać z tą metodą także wszystkie pozostałe kontrolki `TextBlock`, aby była wywoływana również dla nich.

Procedura obsługi zdarzenia jest metodą, którą aplikacja wywołuje w reakcji na zdarzenia takie jak kliknięcie myszą, wciśnięcie klawisza lub zmiana wielkości okna.

Zaostrz ołówek

Oto kod metody `TextBlock_MouseDown`. Zanim dodasz go do programu, przeczytaj go i spróbuj ustalić, co robi. Nie przejmuj się, jeśli nie udzielisz w pełni poprawnych odpowiedzi! Celem jest rozpoczęcie treningu mózgu w rozpoznawaniu kodu C# jako czegoś, co potrafisz odczytać i zrozumieć.

```
TextBlock lastTextBlockClicked;
bool findingMatch = false;

private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
    TextBlock textBlock = sender as TextBlock;
    if (findingMatch == false)
    {
        textBlock.Visibility = Visibility.Hidden;
        lastTextBlockClicked = textBlock;
        findingMatch = true;
    }
    else if (textBlock.Text == lastTextBlockClicked.Text)
    {
        textBlock.Visibility = Visibility.Hidden;
        findingMatch = false;
    }
    else
    {
        lastTextBlockClicked.Visibility = Visibility.Visible;
        findingMatch = false;
    }
}
```

1. Co robi `findingMatch`?

_____
_____

2. Co robi blok kodu zaczynający się od `if (findingMatch == false)`?

_____
_____

3. Co robi blok kodu zaczynający się od `if (textBlock.Text == lastTextBlockClicked.Text)`?

_____
_____

4. Co robi blok kodu zaczynający się od `else`?

_____
_____

Zaostrz ołówek
Rozwiązanie

Oto kod metody `TextBlock_MouseDown`. Zanim dodasz go do programu, przeczytaj go i spróbuj ustalić, co robi. Nie przejmuj się, jeśli nie udzielisz w pełni poprawnych odpowiedzi! Celem jest rozpoczęcie treningu mózgu w rozpoznawaniu kodu C# jako czegoś, co potrafisz odczytać i zrozumieć.

```
TextBlock lastTextBlockClicked;
bool findingMatch = false;

private void TextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
    TextBlock textBlock = sender as TextBlock;
    if (findingMatch == false)
    {
        textBlock.Visibility = Visibility.Hidden;
        lastTextBlockClicked = textBlock;
        findingMatch = true;
    }
    else if (textBlock.Text == lastTextBlockClicked.Text)
    {
        textBlock.Visibility = Visibility.Hidden;
        findingMatch = false;
    }
    else
    {
        lastTextBlockClicked.Visibility = Visibility.Visible;
        findingMatch = false;
    }
}
```

Oto opis działania kodu metody `TextBlock_MouseDown`. Czytanie kodu w nowym języku programowania przypomina czytanie nut. Ta umiejętność wymaga praktyki, a im częściej będziesz ją wykorzystywać, tym lepiej będzie Ci szło.

1. Co robi `findingMatch`?

Rejestruje, czy gracz kliknął już wcześniej pierwsze zwierzę w parze i teraz próbuje znaleźć pasujący rysunek.

2. Co robi blok kodu zaczynający się od `if (findingMatch == false)`?

Gracz kliknął pierwsze zwierzę w parze, dlatego kod ukrywa rysunek tego zwierzęcia i zapisuje powiązaną kontrolkę TextBlock na wypadek, gdyby trzeba było ponownie wyświetlić grafikę.

3. Co robi blok kodu zaczynający się od `if (textBlock.Text == lastTextBlockClicked.Text)`?

Gracz znalazł pasujące zwierzę! Kod ukrywa drugie zwierzę w parze (i uniemożliwia jego kliknięcie) oraz resetuje wartość findingMatch, aby następne kliknięte zwierzę znów było traktowane jak pierwsze w parze.

4. Co robi blok kodu zaczynający się od `else`?

Gracz kliknął zwierzę, które nie pasuje do pierwszego. Dlatego kod ponownie wyświetla pierwsze kliknięte zwierzę i resetuje wartość findingMatch.

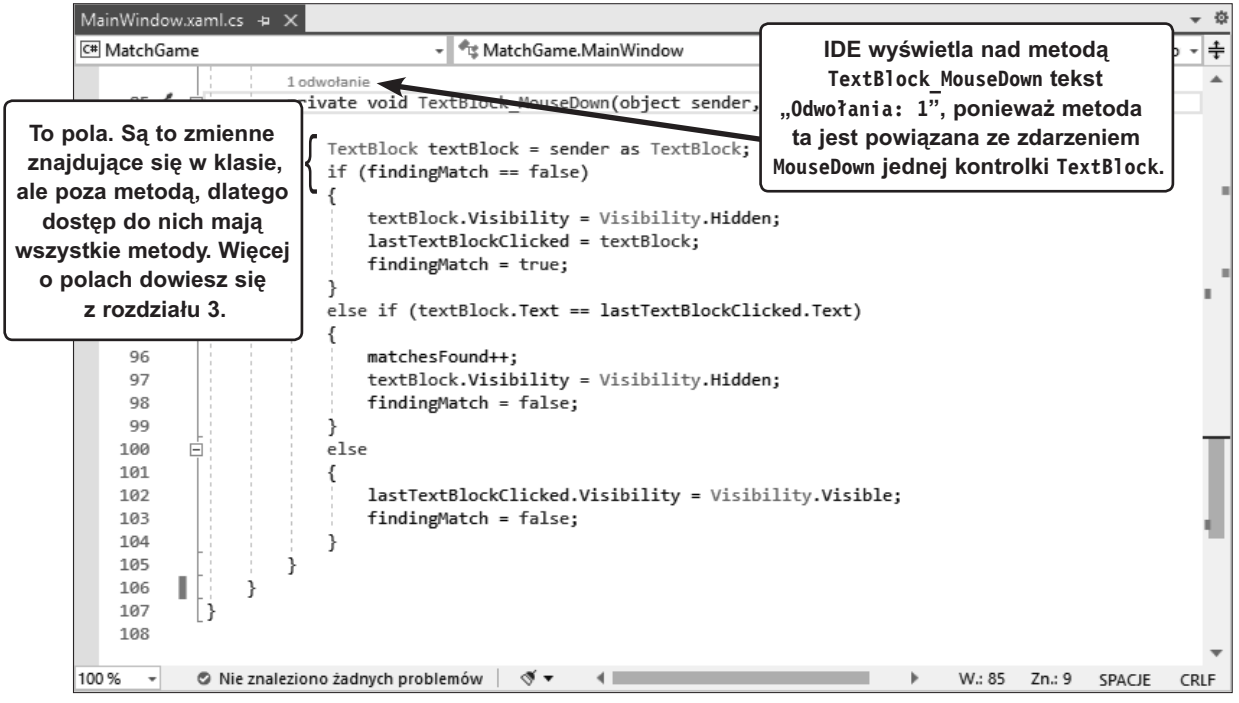
Dodaj kod metody TextBlock_MouseDown

Po zapoznaniu się z kodem metody `TextBlock_MouseDown` pora dodać go do programu. Oto, co powinieneś zrobić:

1. Dodaj dwa pierwsze wiersze z polami `lastTextBlockClicked` i `findingMatch` **nad pierwszym wierszem** metody `TextBlock_MouseDown` dodanej przez IDE. Upewnij się, że znajdują się one między zamykającym nawiasem klamrowym metody `SetUpGame` a nowym kodem dodanym przez IDE.

2. **Uzupełnij kod** metody `TextBlock_MouseDown`. Uważaj na znaki równości — jest duża różnica między symbolami = i == (dowiesz się o niej z następnego rozdziału).

W IDE kod powinien wyglądać tak:



Spraw, aby pozostałe kontrolki TextBlock wywoływały tę samą procedurę obsługi zdarzeń MouseDown

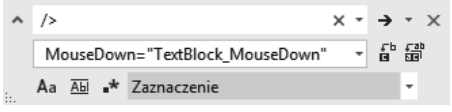
Obecnie tylko pierwsza kontrolka `TextBlock` ma procedurę obsługi zdarzenia `MouseDown`. Pora powiązać z tą procedurą pozostałych 15 kontrolek `TextBlock`. *Mógłbyś* to zrobić, wybierając każdą kontrolkę w *Projektancie* i wpisując `TextBlock_MouseDown` w polu obok zdarzenia `MouseDown`. Wiesz już jednak, że powoduje to tylko dodanie właściwości w kodzie XAML, dlatego można wybrać krótsze rozwiązanie.

1 **Zaznacz pozostałych 15 kontrolek TextBlock w edytorze kodu XAML.**

Przejdź do edytora kodu XAML, kliknij po lewej stronie drugiego znacznika `TextBlock` i przeciągnij kursor w dół do końca listy kontrolek `TextBlock` (do miejsca przed końcowym znacznikiem `</Grid>`). Teraz zaznaczonych powinno być ostatnich 15 kontrolek (bez pierwszej).

2 **Użyj funkcji Szybkie zamienianie, aby dodać procedury obsługi zdarzenia MouseDown.**

Wybierz opcję ***Znajdź i zamień*/*Szybkie zamienianie*** w menu *Edycja*. Jako szukaną sekwencję wpisz `/>` i zastąp ją kodem `MouseDown="TextBlock_MouseDown"/>`. Pamiętaj, aby przed kodem `MouseDown` znajdowała się spacja, a jako zakres wyszukiwania wpisz ***Zaznaczenie*** (właściwość zostanie więc dodana tylko do zaznaczonych kontrolek `TextBlock`).



Przed MouseDown znajduje się spacja, aby nowa właściwość nie została połączona z wcześniejszą.

3 **Zamień tekst we wszystkich 15 zaznaczonych kontrolkach TextBlock.**

Kliknij przycisk *Zamień wszystkie* (), aby dodać właściwość `MouseDown` do kontrolek `TextBlock`. Narzędzie powinno poinformować o 15 operacjach zamiany. Starannie sprawdź kod XAML, aby się upewnić, że każda kontrolka ma właściwość `MouseDown` identyczną z właściwością z pierwszej kontrolki `TextBlock`.

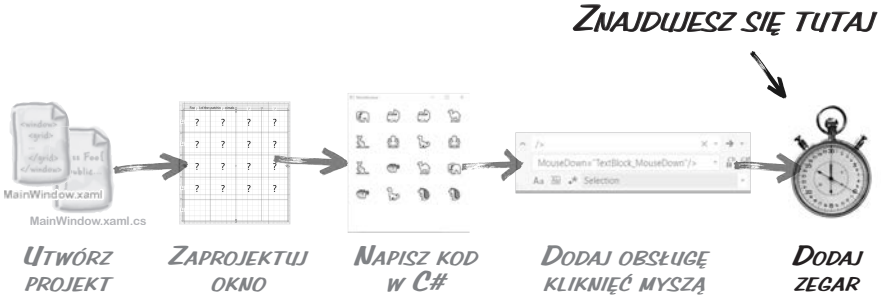
Upewnij się też, że edytor kodu C# informuje o **16 odwołaniach** do metody (wybierz opcję *Kompilowanie*/*Kompiluj rozwiązanie*, aby zaktualizować kod). Jeśli edytor informuje o 17 odwołaniach, przypadkowo dodałeś procedurę obsługi zdarzeń do kontrolki `Grid`. Z pewnością tego nie chcesz, ponieważ kliknięcie zwierzęcia spowoduje wtedy wyjątek.

Uruchom program. Teraz możesz klikać pary zwierząt, aby usunąć je z siatki. Pierwsze kliknięte zwierzę znika. Jeśli potem klikniesz pasujące zwierzę, także ono zniknie. Jednak gdy klikniesz niepasujący rysunek, pierwsze zwierzę znów się pojawi. Po zniknięciu wszystkich zwierząt możesz ponownie uruchomić program lub go zamknąć.

Gdy napotkasz zadanie Wysil szare komórki, zatrzymaj się na chwilę i dobrze się zastanów nad zadanym pytaniem.

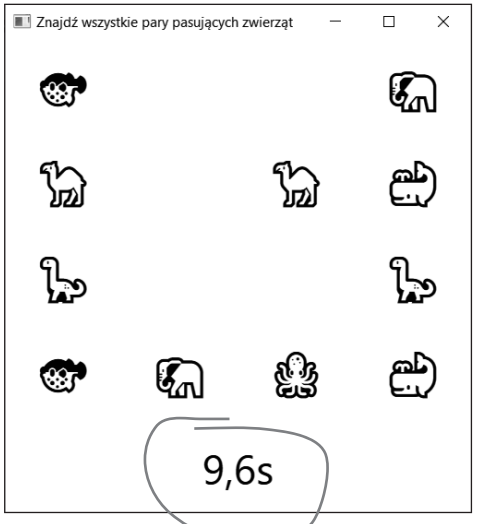
WYSIL SZARE KOMÓRKI

Dotarłeś do punktu kontrolnego w projekcie! Gra jeszcze nie jest skończona, ale działa i można w nią grać. Dlatego jest to dobry moment na zastanowienie się, jak można ją ulepszyć. Co możesz zrobić, aby stała się ciekawsza?



Ukończ grę, dodając zegar

Gra w dopasowywanie zwierząt będzie ciekawsza, jeśli gracze będą mogli próbować poprawić swój najlepszy czas. Dlatego dodasz **zegar**, który będzie „tykał” po określonych przedziałach czasu, wielokrotnie wywołując metodę.



Spraw, by gra stała się ciekawsza! W dolnej części okna ma się pojawiać czas od rozpoczęcia gry. Czas ma stale biec naprzód, a zatrzymywać się dopiero po dopasowaniu ostatniego zwierzęcia.

Zegar „tyka” po upływie określonych przedziałów czasu, w kółko wywołując metodę. Odliczanie ma się zaczynać w momencie rozpoczęcia gry przez gracza i kończyć po dopasowaniu ostatniego zwierzęcia.

Dodaj zegar do kodu gry

Dodaj to!

1. Najpierw znajdź słowo kluczowe `namespace` na początku pliku *MainWindow.xaml.cs* i bezpośrednio pod nim dodaj wiersz `using System.Windows.Threading;`.

```
namespace MatchGame
{
    using System.Windows.Threading;
```

2. Znajdź fragment `public partial class MainWindow` i zaraz po otwierającym nawiasie klamrowym, {, **dodaj ten kod**:

```
public partial class MainWindow : Window
{
    DispatcherTimer timer = new DispatcherTimer();
    int tenthsOfSecondsElapsed;
    int matchesFound;
```

Te trzy wiersze tworzą nowy zegar i dodają dwa pola do śledzenia upływu czasu i liczby dopasowanych par.

3. Trzeba poinstruować zegar, jak często ma „tykać” i jaką metodę ma wywoływać. Kliknij początek wiersza, w którym wywołujesz metodę `SetUpGame`, aby przenieść tam kursor edytora. Wciśnij *Enter*, a następnie wpisz dwa wiersze kodu z poniższego zrzutu zaczynające się od członu **`timer.`**. Zaraz po wpisaniu `+=` IDE wyświetli widoczny komunikat.

```
Odwołania: 0
public MainWindow()
{
    InitializeComponent();

    timer.Interval = TimeSpan.FromSeconds(.1);
    timer.Tick +=
    SetUpGame();
}
```

Timer_Tick1; (Naciśnij klawisz TAB, aby wstawić)

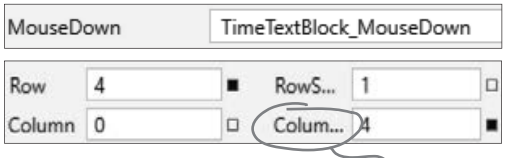
Następnie dodaj te dwa wiersze kodu. Zacznij wpisywać drugi wiersz: „timer.Tick +=”. Zaraz po wpisaniu znaku równości IDE wyświetli komunikat „Naciśnij klawisz TAB, aby wstawić”.

4. Wciśnij klawisz *Tab*. IDE dokończy wiersz kodu i doda metodę `Timer_Tick`:

```
Odwołania: 0
public MainWindow()
{
    InitializeComponent();

    timer.Interval = TimeSpan.FromSeconds(.1);
    timer.Tick += Timer_Tick;
    SetUpGame();
}

1 odwołanie
private void Timer_Tick(object sender, EventArgs e)
{
    throw new NotImplementedException();
}
```

Gdy wciśniesz klawisz Tab, IDE automatycznie wstawi metodę wywoływaną przez zegar.

5 Metoda `Timer_Tick` ma aktualizować kontrolkę `TextBlock` zajmującą cały dolny wiersz siatki. Wykonaj następujące operacje:



Właściwość ColumnSpan znajduje się w sekcji Układ okna Właściwości. Do przełączania się między właściwościami i zdarzeniami użyj przycisków w górnej części okna.

- ★ Przeciągnij kontrolkę **`TextBlock`** w lewe dolne pole.
- ★ Użyj **pola *Name*** w górnej części okna *Właściwości*, aby nazwać kontrolkę **`timeTextBlock`**.
- ★ Zresetuj **marginesy**, **wyśrodkuj** kontrolkę w komórce, ustaw właściwość ***FontSize*** na `36px`, a właściwość ***Text*** na `Czas: ` (podobnie jak robiłeś to w innych kontrolkach).
- ★ Znajdź właściwość ***ColumnSpan*** i ustaw jej wartość na `4`.
- ★ Dodaj **procedurę obsługi zdarzeń *MouseDown*** o nazwie `TimeTextBlock_MouseDown`.

Kod XAML kontrolki powinien wyglądać tak (starannie porównaj ten fragment z kodem w Twoim IDE):

```
<TextBlock x:Name="timeTextBlock" Text="Czas: " FontSize="36"
    HorizontalAlignment="Center" VerticalAlignment="Center"
    Grid.Row="4" Grid.ColumnSpan="4" MouseDown="TimeTextBlock_MouseDown"/>
```

6 Gdy dodajesz procedurę obsługi zdarzeń `MouseDown`, Visual Studio tworzy w kodzie zaplecza metodę `TimeTextBlock_MouseDown` (podobnie jak dla innych kontrolek `TextBlock`). Dodaj do niej ten kod:

```
private void TimeTextBlock_MouseDown(object sender, MouseButtonEventArgs e)
{
    if (matchesFound == 8)
    {
      SetUpGame();
    }
}
```

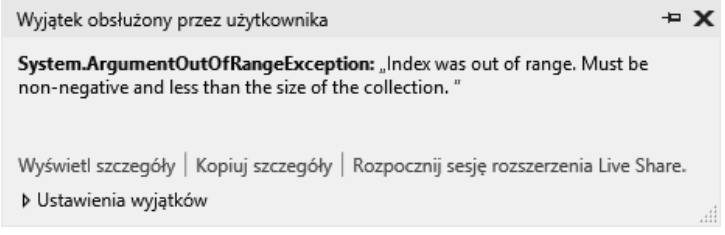
Ten kod resetuje grę, jeśli użytkownik znalazł wszystkich 8 pasujących par (w przeciwnym razie nic nie robi, ponieważ gra wciąż jest w toku).

7 Masz już wszystko, czego potrzebujesz do ukończenia metody `Timer_Tick`, która aktualizuje nową kontrolkę `TextBlock` na podstawie czasu, jaki upłynął, i zatrzymuje zegar po dopasowaniu wszystkich zwierząt:

```
private void Timer_Tick(object sender, EventArgs e)
{
    tenthsOfSecondsElapsed++;
    timeTextBlock.Text = (tenthsOfSecondsElapsed / 10F).ToString("0.0s");
    if (matchesFound == 8)
    {
        timer.Stop();
        timeTextBlock.Text = timeTextBlock.Text + " – Jeszcze raz?";
    }
}
```

Coś jednak jest nie tak. Uruchom kod — ups! Wystąpił **wyjątek**.

Zaraz naprawisz usterkę, jednak najpierw przyjrzyj się komunikatowi o błędzie i wyróżnionemu wierszowi w IDE.



Czy potrafisz odgadnąć, co spowodowało błąd?

O-o! Jak sądzisz, co się stało?

Użyj debugera do znalezienia przyczyny wyjątku

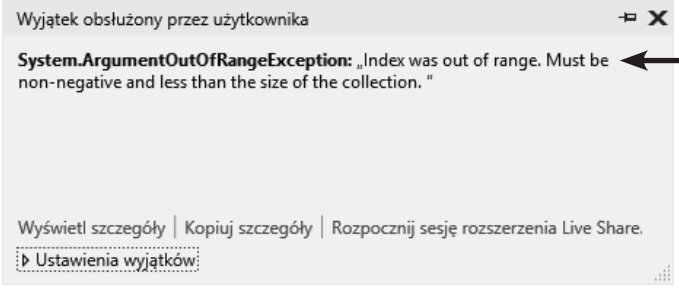
Możliwe, że zetknąłeś się już ze słowem „bug” (dosłownie „robak”, ale wyraz ten oznacza błąd). Może nawet słyszałeś, jak któryś z Twoim znajomych stwierdził: „Ta gra jest pełna bugów, ma mnóstwo usterek”. Każdy błąd ma wyjaśnienie i wszystko w programie dzieje się z jakiegoś powodu, jednak czasem trudno jest ustalić przyczynę usterki.

Zrozumienie błędu jest pierwszym krokiem do jego usunięcia. Na szczęście debuger w Visual Studio jest doskonałym narzędziem, które Ci w tym pomoże. Nazwa debuger pochodzi właśnie od tego, że narzędzie to pomaga eliminować błędy (bugi).

Zdebuguj to!

1 **Kilkakrotnie zrestartuj grę.**

Pierwszą rzeczą, jaką warto zauważyć, jest to, że program zawsze zgłasza ten sam rodzaj wyjątku z tym samym komunikatem:



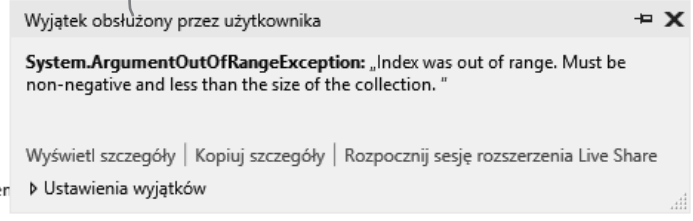
Wyjątki służą językowi C# do informowania o tym, że kod zadziałał nieprawidłowo. Każdy wyjątek jest określonego typu. Tu jest to typ `ArgumentOutOfRangeException`. Z wyjątkami powiązane są przydatne komunikaty pomagające ustalić przyczynę problemu. Tu treść komunikatu to: „Index was out of range”, czyli „parametr index jest poza zakresem”. Jest to cenna informacja pozwalająca stwierdzić, co poszło nie tak.

Gdy zobaczysz wyjątek, często możesz go traktować jak dobrą wiadomość — znalazłeś błąd i możesz go poprawić.

Jeśli przeniesiesz okno wyjątku, zobaczysz, że program zawsze zatrzymuje się w tym samym wierszu:

Ten wiersz powoduje zgłoszenie wyjątku.

```
foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
{
        int index = random.Next(animalEmoji.Count);
        string nextEmoji = animalEmoji[index];
        textBlock.Text = nextEmoji;
        animalEmoji.RemoveAt(index);
}
}

TextBlock lastTextBlockClicked;
bool findingMatch = false;

Odwołania: 16
private void TextBlock_MouseDown(object ser
{
```



Ten wyjątek jest **powtarzalny**: możesz spowodować, by program zgłosił dokładnie ten sam wyjątek, i dobrze wiesz, gdzie znajduje się problem.

Anatomia debugera

Gdy aplikacja zostaje zatrzymana w debugerze (to tak zwane „przerwanie" aplikacji), na pasku narzędzi pojawiają się kontrolki debugera. W trakcie lektury tej książki nabierzesz wprawy w ich używaniu, dlatego nie musisz zapamiętywać, jak działają. Na razie przeczytaj opisy i umieść kursor nad poszczególnymi przyciskami, aby zobaczyć ich nazwy i skróty klawiaturowe.

Możesz użyć przycisku Przerwij wszystkie, aby wstrzymać aplikację. Gdy aplikacja jest już wstrzymana, przycisk jest nieaktywny.

Przycisk Uruchom ponownie restartuje aplikację. Jest to odpowiednik zatrzymania i ponownego uruchomienia aplikacji.

Przycisk Wkrocz do wykonuje następną instrukcję. Jeśli jest to wywołanie metody, wykonywana jest tylko pierwsza instrukcja z danej metody.

Przycisk Przekrocz nad też wykonuje następną instrukcję, jeśli jednak jest to wywołanie metody, wykonywana jest ona w całości.

Ten przycisk wznawia działanie aplikacji. Jeśli wciśniesz go teraz, ponownie zgłoszony zostanie ten sam wyjątek.

Już używałeś przycisku Zatrzymaj debugowanie do zatrzymywania aplikacji.

Przycisk Pokaż następną instrukcję przenosi kursor do następnej instrukcji, która ma zostać wykonana.

Przycisk Wyjdź kończy wykonywanie bieżącej metody i wstrzymuje program w pierwszym wierszu po wywołaniu tej metody.

2 Dodaj punkt przerwania w wierszu, który zgłasza wyjątek.

Ponownie uruchom program, aby zatrzymał się w miejscu wystąpienia wyjątku. Zanim zatrzymasz program, wybierz opcję ***Przełącz punkt przerwania (F9)*** z menu *Debugowanie*. Bieżący wiersz zostanie wtedy wyróżniony czerwonym kolorem, a na lewym marginesie pojawi się czerwona kropka. Teraz możesz **ponownie zatrzymać aplikację**. Wyróżnienie i kropka pozostaną w tym samym miejscu:

```
67
68    int index = random.Next(animalEmoji.Count);
69    string nextEmoji = animalEmoji[index];
      textBlock.Text = nextEmoji;
```

Właśnie dodałeś punkt przerwania do wiersza. Program będzie teraz wstrzymywany za każdym razem, gdy będzie wykonywał dany wiersz. Sprawdź to teraz. Ponownie uruchom aplikację. Program wstrzyma pracę w tym samym wierszu, ale tym razem ***nie zgłosi wyjątku***. Wciśnij przycisk *Kontynuuj*. Program ponownie wstrzyma działania w tym wierszu. Jeszcze raz wciśnij *Kontynuuj*. Program znów wstrzyma pracę. Powtarzaj tę operację do momentu wystąpienia wyjątku. Wtedy zatrzymaj aplikację.

Zaostrz ołówek

Ponownie uruchom aplikację, ale tym razem bacznie ją obserwuj i odpowiedz na następujące pytania:

1. Ile razy aplikacja wstrzymuje pracę w punkcie przerwania przed zgłoszeniem wyjątku? _____

2. W trakcie debugowania aplikacji pojawia się okno *Lokalne*. Jak myślisz, do czego służy to okno? Jeśli go nie widzisz, wybierz w menu opcję ***Debugowanie/Okna/Lokalne (Ctrl+D, L)***.

_____

_____

Zaostrz ołówek
Rozwiązanie

Aplikacja wstrzymała pracę 17 razy. Po 17. razie zgłosiła wyjątek.

Okno Lokalne wyświetla aktualne wartości zmiennych i pól. Możesz go używać do obserwowania zmian tych wartości w trakcie wykonywania programu.

3 **Zbierz dowody, aby ustalić, co powoduje problem.**

Czy zauważyłeś coś ciekawego w oknie *Lokalne* po uruchomieniu aplikacji? Uruchom ją ponownie i bacznie śledź zmienną `animalEmoji`. Gdy aplikacja po raz pierwszy wstrzyma prace, w oknie *Lokalne* powinieneś zobaczyć takie informacje:

Wciśnij przycisk *Kontynuuj*. Wygląda na to, że wartość licznika zmniejszyła się o 1 (z 16 na 15):

Aplikacja dodaje losowe emocji z listy `animalEmoji` do kontrolek `TextBlock`, a następnie usuwa dodane emoji z tej listy, dlatego licznik za każdym razem powinien zmniejszać się o 1. Wszystko idzie dobrze do czasu, gdy lista `animalEmoji` staje się pusta (licznik wtedy przyjmuje wartość 0) i aplikacja zgłasza wyjątek. To pierwszy dowód! Drugim dowodem jest to, że wyjątek jest zgłaszany w **pętli `foreach`**. Ostatnim dowodem jest to, że ***problemy pojawiły się po dodaniu do okna nowej kontrolki `TextBlock`***.

Czas wcielić się w Sherlocka Holmesa. Czy potrafisz wydedukować, co powoduje wyjątek?

Za kulisami

Instrukcja foreach to rodzaj pętli i wykonuje operacje dla każdego elementu kolekcji.

Pętla umożliwia wielokrotne wykonywanie bloku kodu. W pokazanym kodzie używana jest **pętla `foreach`**. Jest to specjalna pętla wykonująca ten sam kod dla każdego elementu kolekcji (na przykład listy `animalEmoji`). Oto przykładowa pętla `foreach` przetwarzająca listę liczb:

```
List<int> numbers = new List<int>() { 2, 5, 9, 11 };
foreach (int aNumber in numbers)
{
    Console.WriteLine("Liczba to: " + aNumber);
}
```

Ta pętla foreach wykonuje instrukcję Console.WriteLine dla każdej wartości z listy liczb całkowitych.

Ta pętla `foreach` tworzy nową zmienną, `aNumber`. Następnie przetwarza po kolei elementy listy `numbers` i dla każdego z nich wywołuje instrukcję `Console.WriteLine`, przypisując do `aNumber` kolejne wartości z listy:

```
Liczba to: 2
Liczba to: 5
Liczba to: 9
Liczba to: 11
```

Pętla foreach wielokrotnie wykonuje ten sam kod dla każdego elementu kolekcji, za każdym razem przypisując do zmiennej następny element. Tu pętla przypisuje do aNumber kolejną liczbę z listy i używa tej wartości do wyświetlenia wiersza tekstu.

Wprowadzamy tu nowe zagadnienie (choć tylko pokrótce), aby działanie kodu było zrozumiałe. Więcej o pętlach dowiesz się z rozdziału 2. Dalej, w rozdziale 3., wrócimy do pętli `foreach`. Napiszesz wtedy pętlę bardzo podobną do tej z tego rozdziału. Dlatego jeśli wydaje Ci się, że omawiamy materiał trochę za szybko, to w trakcie lektury rozdziału 3. będziesz mógł wrócić do tego przykładu i zobaczyć, czy kod stanie się bardziej zrozumiały. Uważamy, że ponowna lektura kodu po lepszym opanowaniu języka bardzo pomaga przyswoić sobie różne techniki. Nie martw się więc, jeśli niektóre kwestie wydają Ci się niejasne.

Wydedukuj to

4 Ustal, co jest przyczyną błędu.

Program przestaje działać, ponieważ próbuje pobrać następne emoji z listy `animalEmoji`, ale ta jest już pusta. To skutkuje wyjątkiem `ArgumentOutOfRange`. Co spowodowało brak emoji do pobrania?

Przed wprowadzeniem najnowszej zmiany program działał. Następnie dodałeś kontrolkę `TextBlock` i wtedy aplikacja przestała funkcjonować — i to w pętli, która iteracyjnie przetwarza wszystkie kontrolki `TextBlock`. To niezwykle ciekawe…

A zatem, kiedy uruchamiasz aplikację, ***wstrzymuje ona pracę w tym wierszu przy przetwarzaniu każdej kontrolki `TextBlock` z okna***. Dla pierwszych 16 kontrolek `TextBlock` kod działa poprawnie, ponieważ w kolekcji dostępne są emoji:

```
foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
{
    int index = random.Next(animalEmoji.Count);
    string nextEmoji = animalEmoji[index];
    textBlock.Text = nextEmoji;
    animalEmoji.RemoveAt(index);
}
```

Debuger wyróżnia instrukcję, którą ma uruchomić. Tak wygląda ona tuż przed zgłoszeniem wyjątku.

Jednak teraz, kiedy w dole okna znajduje się nowa kontrolka `TextBlock`, program wstrzymuje pracę po raz 17., a ponieważ kolekcja `animalEmoji` zawiera tylko 16 emoji, jest już pusta:

Oznacza to, że przed wprowadzeniem zmiany było 16 kontrolek `TextBox` i lista 16 emoji. Liczba emoji była wystarczająca, aby dodać po jednym do każdej kontrolki `TextBlock`. Jednak teraz masz 17 kontrolek `TextBlock` i tylko 16 emoji, dlatego programowi kończą się emoji do dodania, po czym zgłasza wyjątek.

5 Napraw błąd.

Ponieważ wyjątek jest zgłaszany z powodu wyczerpania się listy emoji w pętli, która przetwarza kontrolki `TextBlock`, można rozwiązać problem, pomijając dodaną kontrolkę. W tym celu sprawdź nazwę kontrolki `TextBlock` dodanej na potrzeby wyświetlania czasu i pomiń ją. Usuń punkt przerwania (aby to zrobić, kliknij go ponownie lub wybierz opcję ***Usuń wszystkie punkty przerwania*** (***Ctrl+Shift+F9***) z menu *Debugowanie*).

```
foreach (TextBlock textBlock in mainGrid.Children.OfType<TextBlock>())
{
    if (textBlock.Name != "timeTextBlock")
    {
        textBlock.Visibility = Visibility.Visible;
        int index = random.Next(animalEmoji.Count);
        string nextEmoji = animalEmoji[index];
        textBlock.Text = nextEmoji;
        animalEmoji.RemoveAt(index);
    }
}
```

Dodaj tę instrukcję `if` w pętli `foreach`, aby pomijała kontrolkę `TextBlock` o nazwie `timeTextBlock`.

Dodaj ten kod, aby naprawić błąd.

Nie jest to jedyny sposób na usunięcie błędu. Gdy zaczniesz pisać więcej kodu, zobaczysz, że istnieje wiele, wiele, WIELE możliwości rozwiązania każdego problemu. I błędy nie są tu wyjątkiem (bez żadnych skojarzeń).

Dodaj resztę kodu i dokończ grę

Do zrobienia pozostała jedna rzecz. Metoda `TimeTextBlock_MouseDown` sprawdza wartość pola `matchesFound`, ale ta wartość nie jest nigdzie zmieniana. Dodaj więc trzy pokazane tu wiersze do metody `SetUpGame`; umieść je bezpośrednio po zamykającym nawiasie klamrowym pętli `foreach`:

```
                animalEmoji.RemoveAt(index);
            }
        }

        timer.Start();
        tenthsOfSecondsElapsed = 0;
        matchesFound = 0;
    }
```

Dodaj te trzy wiersze kodu na końcu metody SetUpGame, aby uruchamiać zegar i resetować pola.

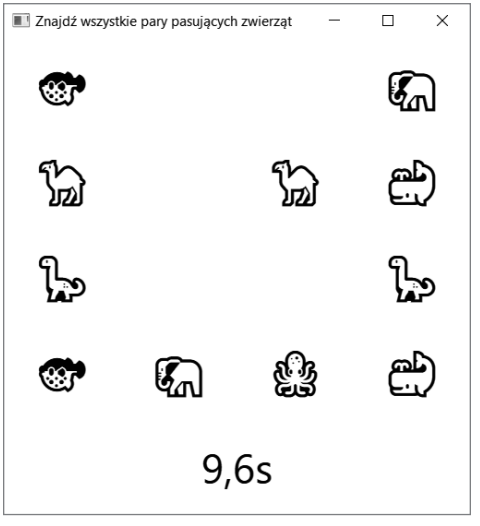
Następnie dodaj poniższą instrukcję wewnątrz bloku `if/else` w metodzie `TextBlock_MouseDown`:

```
else if (textBlock.Text == lastTextBlockClicked.Text)
{
    matchesFound++;
    textBlock.Visibility = Visibility.Hidden;
    findingMatch = false;
}
```

Dodaj ten wiersz kodu, aby zwiększać wartość pola matchesFound o jeden za każdym razem, gdy użytkownik dopasuje zwierzęta.

Teraz w grze działa zegar zatrzymywany, gdy gracz zakończy dopasowywanie zwierząt. Po zakończeniu gry można kliknąć napis „Jeszcze raz?", aby zagrać ponownie.
Napisałeś swoją pierwszą grę w C#. Gratulacje!

Teraz w grze działa zegar informujący, ile czasu graczowi zajęło znalezienie wszystkich par. Czy potrafisz poprawić swój najlepszy czas?



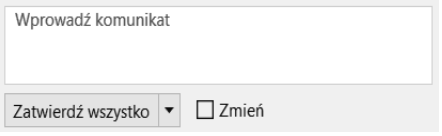
Na stronie https://github.com/head-first-csharp/fourth-edition możesz przejrzeć i pobrać kompletny kod tego projektu, a także wszystkich innych programów z tej książki. Spolszczoną wersję kodu znajdziesz w witrynie wydawnictwa Helion.

Aktualizowanie kodu w systemie kontroli wersji

Gra działa i jest gotowa. Jest to świetny moment, aby **wypchnąć zmiany do systemu Git**. Visual Studio umożliwia łatwe wykonanie tego zadania. Wystarczy *przygotować* zatwierdzenia, wprowadzić komunikat zatwierdzenia, a następnie zsynchronizować projekt ze zdalnym repozytorium.

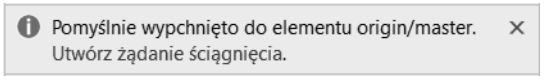
Możesz używać poleceń z menu Git do tworzenia nowych zatwierdzeń z najnowszymi zmianami w kodzie i wypychania ich do repozytorium systemu Git

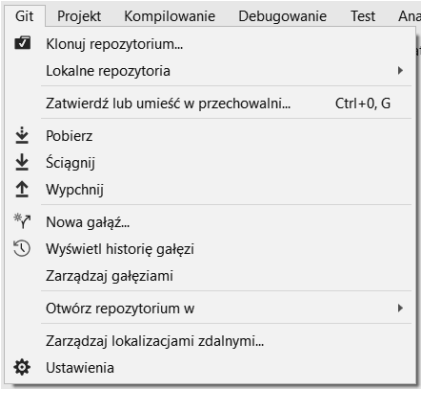
1. Wybierz opcję *Zatwierdź lub umieść w przechowalni (Ctrl+0, G)* z menu *Git*. Wprowadź **komunikat zatwierdzenia** z listą zmian.



2. Wciśnij **przycisk *Zatwierdź wszystko***. Visual Studio wyświetli informację o lokalnym utworzeniu zatwierdzenia.

3. Wybierz opcję ***Wypchnij*** z menu *Git*, aby wypchnąć zatwierdzenie do repozytorium. Po zakończeniu przesyłania kodu pojawi się informujący o tym komunikat.





Przesyłanie kodu do repozytorium w systemie Git jest opcjonalne, ale naprawdę warto to robić.

Przesyłanie kodu do repozytorium w systemie Git jest opcjonalne, ale naprawdę warto to robić.

Rozbicie gry na mniejsze fragmenty, którymi mogłem zająć się pojedynczo, było świetnym pomysłem.

Gdy pracujesz nad dużym projektem, zawsze warto podzielić go na mniejsze części.

Jedna z najprzydatniejszych umiejętności programistycznych dotyczy podziału dużych, skomplikowanych problemów na mniejsze i łatwiejsze fragmenty.

Na początku dużego projektu łatwo dać się przytłoczyć i pomyśleć: „No tak, to prawdziwy gigant”. Jeśli jednak uda Ci się znaleźć mniejszy fragment, nad którym możesz popracować, pozwoli Ci to zacząć. Po ukończeniu tego fragmentu możesz przejść do następnej niewielkiej części, i kolejnej, i jeszcze jednej. W trakcie rozwijania każdego fragmentu coraz lepiej będziesz poznawać cały projekt.

Gra będzie jeszcze lepsza, gdy...

Twoja gra jest całkiem niezła! Ale każdą grę (i prawie każdy program) można ulepszyć. Oto kilka rzeczy, które naszym zdaniem mogłyby pomóc poprawić grę:

- ★ Dodanie różnych gatunków zwierząt, aby nie powtarzać za każdym razem tych samych rysunków.
- ★ Zapisywanie najlepszego czasu gracza, aby mógł próbować go pobić.
- ★ Odliczanie czasu do zera (zamiast od zera), aby gracz miał ograniczoną ilość czasu na wykonanie zadania.

MINI

Zaostrz ołówek

Czy potrafisz wymyślić własne ulepszenia? To świetne ćwiczenie. Zastanów się przez kilka minut i zapisz przynajmniej trzy ulepszenia gry w dopasowywanie zwierząt.

Mówimy poważnie — poświęć kilka minut na wykonanie tego zadania. Zrobienie przerwy w lekturze i zastanowienie się nad ukończonym projektem to świetny sposób na utrwalenie zdobytych informacji.

CELNE SPOSTRZEŻENIA

- Visual Studio śledzi, ile razy metoda jest **wywoływana** w kodzie C# i XAML.
- **Procedura obsługi zdarzeń** to metoda, którą aplikacja wywołuje po zajściu określonego zdarzenia (takiego jak kliknięcie myszą, wciśnięcie klawisza lub zmiana wielkości okna).
- IDE ułatwia **dodawanie i kontrolowanie** procedur obsługi zdarzeń.
- **Okno *Lista błędów*** w IDE wyświetla błędy uniemożliwiające kompilację kodu.
- **Zegary** w kółko wykonują procedury obsługi zdarzeń `Tick` w określonych odstępach czasu.
- Pętla **`foreach`** iteracyjnie przetwarza elementy z kolekcji.
- Gdy program zgłasza **wyjątek**, zbierz dowody i spróbuj ustalić przyczynę problemu.
- Wyjątki są łatwiejsze do wyeliminowania, jeśli są **powtarzalne**.
- Visual Studio bardzo ułatwia używanie **systemu kontroli wersji** do archiwizowania kodu i śledzenia wszystkich wprowadzonych zmian.
- Możesz przesłać kod do **zdalnego repozytorium systemu Git**. Repozytorium z kodem źródłowym wszystkich projektów z tej książki znajduje się w serwisie GitHub.

Krótkie przypomnienie: w tej książce zamiast nazwy Visual Studio często używamy określenia IDE.

Skorowidz

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

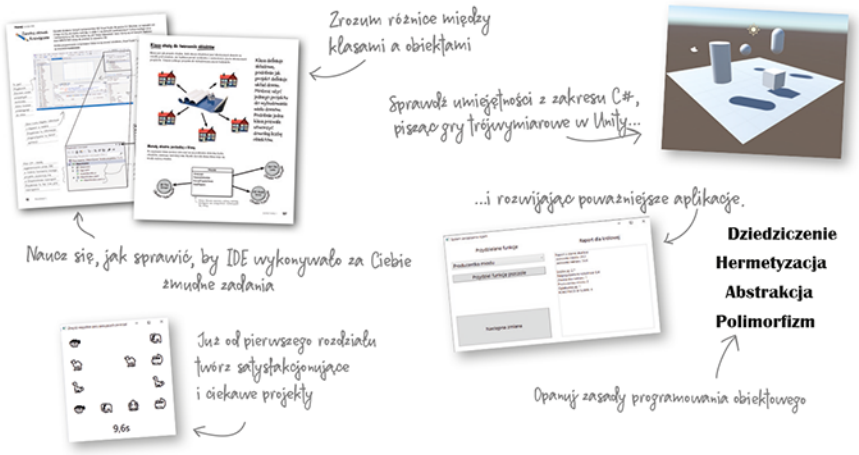
W

X

Z

C#: rusz głową, programuj i ciesz się kodem!

C# jest dziś uważany za jeden z najważniejszych języków programowania. Nowoczesny, wszechstronny, dojrzały i sukcesywnie rozwijany, zapewnia efektywne tworzenie kodu wysokiej jakości. Profesjonalni programiści, którzy pisali już kod w C++ lub Javie, bardzo często wybierają właśnie C#. Nic nie stoi na przeszkodzie, aby był pierwszym językiem programowania przyszłego programisty, który przy okazji nauki chce rozwijać pasję i świetnie się bawić!



Ta książka, podobnie jak inne pozycje z serii *Rusz głową!*, została przygotowana zgodnie z najnowszymi odkryciami nauk poznawczych, teorii uczenia się i neurofizjologii. Oznacza to tyle, że dzięki niej zaangażujesz swój mózg, wykorzystasz wiele zmysłów i niepostrzeżenie przyswoisz język C# 8.0 i umiejętność pracy w Visual Studio 2019. Naukę programowania rozpoczniesz od napisania działającej gry. Później dowiesz się, jak używać klas, czym jest programowanie obiektowe, jak tworzyć gry trójwymiarowe w Unity i jak korzystać z technologii LINQ. Będziesz się tego uczyć, rozwiązując łamigłówki, wykonując praktyczne ćwiczenia i pisząc aplikacje. Ani się obejrzysz, a staniesz się znakomitym, gotowym na dalsze wyzwania programistą C#!

Wielkie dzięki! Wasze książki pomogły mi w rozpoczęciu kariery

– Ryan White
programista gier

Andrew i Jennifer napisali zwięzłe, rzetelne, a przede wszystkim ciekawe wprowadzenie do programowania w C#

– Jon Galloway
starszy menedżer programu w zespole .NET Community, Microsoft

Jeśli chcesz dokładnie poznać C# i mieć z tego przyjemność, to trzymasz w ręku właściwą książkę

– Andy Parker
początkujący programista C#

Andrew Stellman pisał oprogramowanie dla finansistów, był wiceprezesem banku i zarządzał zespołami programistów.

Jennifer Greene zajmowała się jakością oprogramowania i prowadziła ciekawe projekty. Oboje są inżynierami, konsultantami i autorami książek. Współpracują od lat: piszą o programowaniu, doradzają firmom i występują na konferencjach dla inżynierów, architektów i menedżerów.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-749-8
9 788383 227498
Cena: 169,00 zł