

Materiały do przedmiotu:

Podstawy programowania

Laboratorium nr 7

Podstawy programowania obiektowego

**Tworzenie klas i obiektów,
definiowanie właściwości i metod**

Autor:

dr Michał Dolecki



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Spis treści

Wprowadzenie, tematyka zajęć	3
Cel laboratorium	6
Instrukcja laboratoryjna	6
Ćwiczenia samodzielne	18
Pytania pomocnicze	23
Podsumowanie	23
Literatura	23



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wprowadzenie, tematyka zajęć

Programowanie obiektowe (ang. object-oriented programming, OOP) to jeden z najważniejszych i najczęściej stosowanych paradygmatów współczesnego programowania. Paradygmat to zbiór reguł, konwencji i technik uznawanych za najlepszy sposób rozwiązywania określonej klasy problemów. W kontekście programowania możemy wyróżnić m.in. paradygmat strukturalny, proceduralny oraz obiektowy. Choć idee programowania obiektowego sięgają lat 60. XX wieku, prawdziwy przełom w jego popularyzacji nastąpił w połowie lat 90., gdy firma Sun Microsystems wprowadziła język Java. Jego sukces szybko zwrócił uwagę konkurencji. W odpowiedzi Microsoft opracował własne, kompletne środowisko: język C# i platformę .NET, które zadebiutowały na przełomie XX i XXI wieku. Od tego czasu C# wraz z .NET nieprzerwanie należą do czołówki technologii stosowanych w profesjonalnym oprogramowaniu – zarówno na serwerach, desktopie, jak i w chmurze czy na urządzeniach mobilnych.

Zalety programowania obiektowego

1. Modularność kodu

Podstawową zaletą programowania obiektowego jest modularność – możliwość podziału kodu na niezależne, samodzielne fragmenty (klasy), nad którymi można pracować osobno. Każda klasa odpowiada za konkretny fragment logiki, co ułatwia zarówno rozwój, jak i późniejsze modyfikacje. Tak zorganizowany kod pozwala również na łatwe dzielenie pracy w zespole – różni programiści mogą jednocześnie rozwijać różne moduły bez ryzyka konfliktów.

2. Łatwość testowania

Dzięki modularności każdą klasę lub grupę klas można testować niezależnie. Pozwala to szybko wykrywać błędy i kontrolować poprawność działania poszczególnych komponentów, zanim zostaną one zintegrowane w większą całość.

3. Łatwość modyfikacji i rozbudowy

Obiektowa struktura kodu umożliwia szybkie wprowadzanie zmian bez wpływu na resztę aplikacji. Wystarczy zmodyfikować jedną klasę lub dodać nową, aby rozszerzyć funkcjonalność. Taka elastyczność jest szczególnie istotna w metodykach zwinnych (Agile), gdzie wymagania mogą się zmieniać w trakcie trwania projektu. Jak piszą współautorzy w książce Analiza i projektowanie obiektowe – jedynym pewnikiem w projektowaniu obiektowym jest zmiana.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



4. Wielokrotne wykorzystanie kodu (code reusability)

Jedną z kluczowych zalet OOP jest możliwość ponownego wykorzystania istniejącego kodu. Dobrze zaprojektowane klasy mogą być wykorzystane w wielu projektach – wystarczy je zaimportować lub odziedziczyć. Przyspiesza to proces deweloperski i obniża koszty utrzymania oprogramowania. Firmy z dojrzałymi zespołami często budują własne biblioteki klas, które są wykorzystywane w kolejnych wdrożeniach.

5. Czytelność kodu

Programowanie obiektowe pozwala modelować rzeczywiste problemy w sposób naturalny i zrozumiały. Obiekty w kodzie mogą odpowiadać konkretnym bytom z dziedziny problemu – np. Customer, Invoice, Product. Dzięki temu kod staje się czytelny nie tylko dla programistów, ale także dla analityków czy klientów. Implementacja szczegółów jest ukryta wewnątrz klas, a interakcje odbywają się za pomocą metod o nazwach opisujących działania – np. CalculateDiscount(), SendEmail(), ChangeStatus().

Filary programowania obiektowego

W literaturze spotyka się warianty trzy- lub czteroelementowe. Trzy „filary”, które łatwiej zapamiętać to:

- **enkapsulacja (hermetyzacja),**
- **dziedziczenie,**
- **polimorfizm,**

ale pełnego obrazu warto dodać czwarte, równie podstawowe pojęcie:

- **abstrakcja.**

Abstrakcja

1. Klasa jako szablon

Aby utworzyć obiekty, potrzebujemy matrycy – klasy. Klasa jest abstrakcyjnym opisem wszystkich cech i zachowań wspólnych dla danej kategorii obiektów.

2. Wybór istotnych cech

Abstrakcja to także świadome odrzucenie nieistotnych szczegółów.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład: rzeczywisty student ma setki cech – od numeru buta po kolor oczu. W programie dla klasy Student wybieramy tylko te, które są istotne dla rozwiązywanego problemu:

- Dla dziekanatu: imię, nazwisko, data urodzenia, adres, e-mail, nr telefonu.
- Dla sekcji AZS: dodatkowo wzrost, waga – bo mają wpływ na przydział do drużyn.
- Numer buta? Prawdopodobnie nigdzie nie będzie potrzebny.

Dzięki abstrakcji kod jest zwężyły, czytelny i łatwiejszy w utrzymaniu – modelujemy tylko tę część rzeczywistości, która naprawdę ma znaczenie biznesowe.

Enkapsulacja (hermetyzacja)

Kolejnym filarem jest enkapsulacja, czyli łączenie danych (pól) i operacji (metod) w jedną, zwartą całość – obiekt. Taka „kapsułka” istnieje tylko w paradygmacie obiektowym; w programowaniu proceduralnym dane i funkcje są rozłączne.

Enkapsulacja ma dwa wymiary:

1. fizyczne połączenie – pole i metoda znajdują się w jednej klasie,
2. kontrola dostępu – możliwość ukrycia wybranych składowych przed kodem zewnętrznym.

Sama obecność pól i metod w klasie to enkapsulacja; ukrywanie danych (data hiding) jest jej opcjonalnym, choć bardzo pożądanym efektem.

Przykład: Python pozwala tworzyć klasy z polami i metodami, ale nie wymusza ukrywania – wszystko jest „publiczne”. C# i Java dają narzędzia (modyfikatory dostępu private, protected, public), by zablokować bezpośredni dostęp do pól.

Dlaczego to istotne?

Wróćmy do obiektu klasy Student. Jeśli pole średnia jest publiczne, każdy fragment programu może wpisać tam wartość -17 – oczywisty absurd.

Z ukryciem danych pole staje się private, a jedyną drogą jego zmiany jest metoda setAvg(double avg). W jej ciele sprawdzamy zakres [0–5,0], co chroni obiekt przed błędnym stanem.

Enkapsulacja + ukrywanie danych = bezpieczny i łatwy do utrzymania kod



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Dziedziczenie

Dziedziczenie to związek między dwiema klasami, w którym jedna jest klasą nadrzędną (bazową albo klasą rodzica, z której dziedziczymy), a druga jest klasą podrzędną (potomną, która dziedziczy). Dzięki temu mechanizmowi klasa potomna automatycznie otrzymuje wszystkie nieprywatne składowe klasy bazowej i może je wykorzystywać takie jakie są albo rozszerzać dodając coś nowego lub zupełnie modyfikować.

Polimorfizm

Polimorfizm (z gr. wiele postaci) oznacza, że metoda o tej samej nazwie może zachowywać się inaczej w zależności od klasy, w której jest umieszczona. Dzięki dziedziczeniu klasa bazowa określa nagłówek (sygnaturę) metody, a klasy potomne dostarczają własnej, dopasowanej implementacji, bądź wykorzystują oryginalną postać tej metody.

Cel laboratorium

Celem zajęć jest zapoznanie studentów z praktycznymi aspektami podstawowych pojęć programowania zorientowanego obiektowo w języku C#. Ponadto w obszarze umiejętności celem jest rozwój analitycznego myślenia i rozwiązywania problemów z wykorzystaniem klas i obiektów oraz ćwiczenie stosowania języka C#.

Instrukcja laboratoryjna

Obiekt programistyczny

W podejściu obiektowym skupiamy się na tworzeniu obiektów programistycznych – jednostek łączących w sobie zarówno dane (stan), jak i operacje, które można na tych danych wykonać (zachowanie). Obiekt zawiera więc:

- Dane – opisujące jego aktualny stan (np. imię i wiek osoby),
- Metody – funkcje, które definiują, co dany obiekt może robić lub jakie operacje może wykonywać na innych danych.



Fundusze Europejskie
dla Rozwoju Społecznego

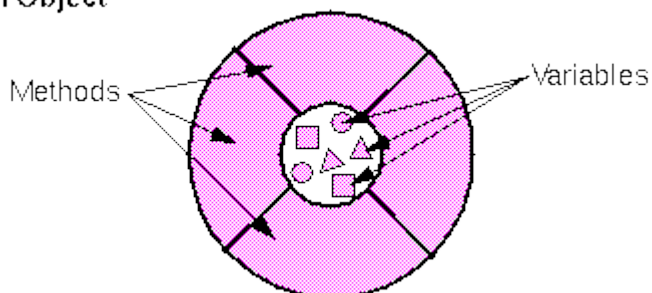


Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



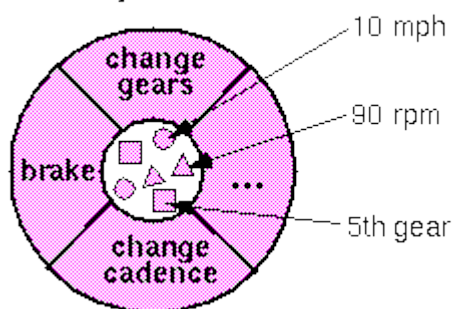
An Object



Rysunek 1. Obiekt programistyczny.

Rysunek 1 przedstawia ogólny obiekt programistyczny. Warto zwrócić uwagę na umieszczenie pól wewnątrz i odizolowanie ich od świata zewnętrznego za pomocą metod, co realizuje hermetyzację i data hiding.

Your Bicycle



Rysunek 2. Obiekt Bicycle.

Na rysunku 2 mamy konkretny obiekt programistyczny YourBicycle utworzony na podstawie klasy Bicycle z konkretnymi wartościami pól i metodami do ich zmiany.

Źródło obu rysunków:

https://web.mit.edu/java_v1.0.2/www/tutorial/java/objects/object.html

W przeciwieństwie do programowania proceduralnego, gdzie główną jednostką jest funkcja, w OOP podstawową jednostką jest obiekt. Program tworzony w tym paradygmacie składa się z wielu obiektów, które ze sobą współpracują, przekazują informacje i wywołują nawzajem swoje metody. Obiekty są tworzone na podstawie klas, które pełnią rolę szablonów lub planów. Klasa definiuje, jakie właściwości i metody będą miały obiekty z niej utworzone.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład 7.1. Podstawowa klasa

Zacznijmy od prostej klasy Car, która jest wzorcem do stworzenia obiektów takich jak myCar, myToyota itd. i niech posiada pola makra (ang. make) i kolor (ang. color) będące napisami. W C# nie jest to wymagane, ale jest pewną konwencją, że używamy PascalCase dla nazw klas.

```
class Car
{
    string make;
    string color = "red";
}
```

Natomiast w głównej metodzie programu możemy utworzyć jeden albo kilka obiektów i pracować na nich uzyskując dostęp do składowych za pomocą operatora „.”.

```
static void Main(string[] args)
{
    Car myObj = new Car();
    myObj.make = "Volvo";
    Console.WriteLine(myObj.color);

    Car myCar = new Car();
}
```

Przykład 7.2. Program jako klasa

Warto zauważyć, że w C# cały główny program jest umieszczony w klasie, więc można poprzedni przykład złączyć w jeden plik i wyglądałoby to następująco:

```
using System;

namespace MyApplication
{
    class Car
    {
        string make;
        string color = "red";

        static void Main(string[] args)
        {
            Car myObj = new Car();
            myObj.make = "Volvo";
            Console.WriteLine(myObj.color);
        }
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską




```
}  
}  
}
```

Głównym elementem tej klasy jest metoda statyczna o nazwie Main, która jako parametry przyjmuje tablicę napisów args. Trafiają do niej parametry, które mogą być podane w konsoli/wierszu poleceń/terminalu podczas uruchamiania programu. Umieszczenie całego kodu w jednej klasie jest możliwe, ale takie rozwiązanie nie jest zalecane dla większych projektów, w których lepiej umieszczać klasy w osobnych plikach. Dzięki temu wykorzystamy lepiej modułowość w OOP. Dla ułatwienia pliki te nazywamy tak jak znajdujące się w nich klasy, chociaż nie jest to wymogiem C# (w odróżnieniu od np. Javy). Mimo, że w dalszej części skryptu będziemy rozdzielać kod klas na pliki, to warto chociaż raz zobaczyć taką podstawową konstrukcję.

Przykład 7.3. Klasy w osobnych plikach

Tworząc poważniejszy projekt należałoby podzielić implementację na osobne pliki. Organizuje to projekt, ułatwia nawigację między fragmentami aplikacji, pozwala podzielić pracę między kilku programistów, a w razie problemów, pozwala na wyłączenie pojedynczego pliku z projektu i osobne jego przetestowanie. Zwiększa to jego czytelność i ułatwia codzienną pracę z kodem.

Plik Car.cs

```
using System;  
  
namespace MyApplication  
{  
    class Car  
    {  
        public string color = "red";  
    }  
}
```

oraz plik App.cs z główną metodą Main:

```
using System;  
  
namespace MyApplication  
{  
    class App  
    {
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
static void Main(string[] args)
{
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
}
}
```

Warto zwrócić uwagę, że w obu plikach jest ta sama przestrzeń nazw.

Przykład 7.4. Pola i metody w klasach

Poza polami klasa może zawierać metody, a więc funkcje wewnątrz klasy. Plik z taką klasą może wyglądać następująco:

```
class Car
{
    public string model;
    public string color;
    public int year;

    public void fullThrottle()
    {
        Console.WriteLine("The car is going as fast as it can!");
    }
}
```

Natomiast główna klasa programu:

```
class App
{
    static void Main(string[] args)
    {
        Car Ford = new Car();
        Ford.model = "Mustang";
        Ford.color = "red";
        Ford.year = 1969;
        Car Opel = new Car();
        Opel.model = "Astra";
        Opel.color = "white";
        Opel.year = 2005;
        Console.WriteLine(Ford.model);
        Console.WriteLine(Opel.model);
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Inicjalizacja pól w zwykłych przypisaniach zajmuje dużo miejsca i pisanie tego zajmuje dużo czasu, dodatkowo jest to kod, który się powtarza więc aż prosi się, aby ująć to w metodę [o czym za moment]. Dodatkowo korzystamy tutaj z posiadania dostępu bezpośrednio do pól, co nie jest bezpieczne i łamie zasady data hiding. Wprowadzimy sobie również modyfikatory dostępu, które uniemożliwiają takie pisanie.

Przykład 7.5. Konstruktor

Wspomniana powyżej metoda do inicjalizacji pól klasy nazywa się konstruktorem i jest to charakterystyczna metoda, która nie ma typu zwracanego oraz ma nazwę taką samą jak nazwa klasy. Wyróżnić możemy konstruktor bezparametrowy i konstruktor z parametrami

Konstruktor bezparametrowy

Ten typ konstruktora pozwala na inicjowanie pól wartościami domyślnymi, które może wybrać programista.

```
class Car
{
    public string model;

    public Car()
    {
        model = "Mustang";
    }
}
```

Każdy utworzony w głównym programie samochód będzie domyślnie Mustangiem.

Konstruktor z parametrami

W sytuacji, kiedy chcemy określać wartości pól w momencie tworzenia obiektów możemy użyć konstruktora parametrowego:

```
class Car
{
    public string model;
    public string color;
    public int year;

    public Car(string modelName, string modelColor, int modelYear)
    {
        model = modelName;
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```

        color = modelColor;
        year = modelYear;
    }
}

```

Metoda Main wyglądałaby teraz następująco:

```

class App
{
    static void Main(string[] args)
    {
        Car Ford = new Car("Mustang", "red", 1969);
        Car Opel = new Car("Astra", "white", 2005);

        Console.WriteLine(Ford.model);
        Console.WriteLine(Opel.model);
    }
}

```

Kod stał się bardziej przejrzysty, zwięzły.

Przykład 7.6. Modyfikatory dostępu

We wszystkich poprzednich przykładach korzystaliśmy z bezpośredniego dostępu do klas, co nie jest zbyt bezpieczne, ale konieczne było omówienie ważniejszych elementów takich jak struktura projektu, podział kodu na osobne klasy i pliki, tworzenie pól i metod klas oraz tworzenie obiektów w głównym programie. Aby wprowadzić praktyczną realizację data hiding dodajemy do każdej składowej modyfikator dostępu. Podstawowe modyfikatory:

Tabela 1. Modyfikatory dostępu

Modyfikator	Opis
public	Kod jest dostępny dla wszystkich klas
private	Kod jest dostępny tylko w obrębie tej samej klasy
protected	Kod jest dostępny w obrębie tej samej klasy lub w klasie dziedziczonej z tej klasy

Dodając modyfikator ograniczający dostęp do pola należy dodać odpowiednie metody ustawiające jego wartość. W poniższym kodzie zostało to zaimplementowane, chociaż nie jest to najlepsze metoda. Niemniej przynajmniej raz warto zobaczyć i takie podejście.

```

public class Car
{
    private string model;
    private string color;
}

```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
private int year;

public string GetModel()
{
    return model;
}

public void SetModel(string value)
{
    model = value;
}

public string GetColor()
{
    return color;
}

public void SetColor(string value)
{
    color = value;
}

public int GetYear()
{
    return year;
}

public void SetYear(int value)
{
    if (value >= 1886 && value <= DateTime.Now.Year + 1)
    {
        year = value;
    }
    else
    {
        throw new ArgumentException("Nieprawidłowy rok produkcji");
    }
}

public Car(string model, string color, int year)
{
    SetModel(model);
    SetColor(color);
    SetYear(year);
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
}  
  
public void DisplayInfo()  
{  
    Console.WriteLine($"Samochód: {GetModel()}, Kolor: {GetColor()},  
Rok: {GetYear()}");  
}  
}
```

Warto zwrócić uwagę na kilka fragmentów tej implementacji. Metoda `SetYear` posiada fragment sprawdzający poprawność przekazanych parametrów. Jeżeli użytkownik podał rok spoza założonego przedziału, metoda przerywa działanie rzucając wyjątek (o tym mechanizmie powiemy sobie później, a na tym etapie wystarczy pamiętać, że wykonanie kodu zostanie przerwane). Konstruktor wykorzystuje metody `Set` do ustawienia wartości pól, pozwala to skrócić implementację ponieważ nie powtarzamy ponownego sprawdzania poprawności pól. W tym przykładzie znajduje się ono tylko w jednym polu, ale łatwo wyobrazić sobie klasy z wieloma polami, które mają wymagane wartości i trzeba je sprawdzić. Dodatkowo w przyszłości, nasza implementacja jest łatwiejsza do zmiany, ponieważ jeżeli zmienią się ograniczenia dla pola, wystarczy poprawić je w jednym miejscu, a nie w kilku. W głównym programie normalnie wywołujemy metody `Get` i `Set` dla obiektu.

Przykład 7.7. Właściwości

Po przeanalizowaniu poprzedniego kodu, można zauważyć, że metody `Get` i `Set` są bardzo powtarzalne, a sam kod stał się bardzo rozwlekły. C# umożliwia skrócenie tej implementacji wprowadzając tak zwane właściwości (ang. properties). Do pól prywatnych dodawane są publiczne właściwości. Dla łatwiejszego zrozumienia kodu dobrze jest stosować dla właściwości te same nazwy, ale pisane wielką literą.

```
public class Car  
{  
    private string model;  
    private string color;  
    private int year;  
  
    public string Model  
    {  
        get { return model; }  
        set { model = value; }  
    }  
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
public string Color
{
    get { return color; }
    set { color = value; }
}

public int Year
{
    get { return year; }
    set
    {
        if (value >= 1886 && value <= DateTime.Now.Year + 1)
        {
            year = value;
        }
        else
        {
            throw new ArgumentException("Nieprawidłowy rok produkcji");
        }
    }
}

public Car(string model, string color, int year)
{
    SetModel(model);
    SetColor(color);
    SetYear(year);
}

public void DisplayInfo()
{
    Console.WriteLine($"Samochód: {Model}, Kolor: {Color}, Rok:
{Year}");
}
}
```

Ważna zmiana zachodzi w głównym programie, gdzie korzystamy z właściwości podając ich nazwy.

```
class App
{
    static void Main(string[] args)
    {
        Car myCar = new Car("Toyota Corolla", "Czerwony", 2020);
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
Console.WriteLine($"Model: {myCar.Model}");  
Console.WriteLine($"Kolor: {myCar.Color}");  
Console.WriteLine($"Rok: {myCar.Year}");  
  
myCar.Color = "Niebieski";  
myCar.Year = 2022;  
  
myCar.DisplayInfo();  
  
try  
{  
    myCar.Year = 1800;  
}  
catch (ArgumentException ex)  
{  
    Console.WriteLine($"Błąd: {ex.Message}");  
}  
}
```

Po wywołaniu metody `DisplayInfo` pokazana została obsługa wyjątku w sytuacji, gdy podano zły rok produkcji. Wykonanie metody `set` zostanie przerwane, wygenerowany zostanie obiekt klasy wyjątku (tu `ArgumentException`), który można przechwycić i obsłużyć informując użytkownika o powodzie przerwania.

Przykład 7.8. Właściwości automatyczne

Powyższy kod klasy jest bardziej zwięzły, ale wciąż zwykłe implementacje `get` i `set` są niemalże identyczne. Tylko `set` ze sprawdzeniem poprawności parametru jest inny. W takiej sytuacji, kiedy nie implementujemy żadnej nietypowej logiki w `get` i `set`, możemy użyć mechanizmu auto-properties, w którym kompilator automatycznie wygeneruje ich kod również z polami prywatnymi (!).

```
public class Car  
{  
    public string Model { get; set; }  
    public string Color { get; set; }  
  
    private int year;  
    public int Year  
    {  
        get { return year; }  
    }  
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską




```
set
{
    if (value >= 1886 && value <= DateTime.Now.Year + 1)
    {
        year = value;
    }
    else
    {
        throw new ArgumentException("Nieprawidłowy rok produkcji");
    }
}

// Konstruktor
public Car(string model, string color, int year)
{
    Model = model;
    Color = color;
    Year = year;
}

public void DisplayInfo()
{
    Console.WriteLine($"Samochód: {Model}, Kolor: {Color}, Rok:
{Year}");
}
```

Na koniec omawiania tej sekcji warto zapytać czy jest potrzeba pisanie w kółko get i set? Może i to zautomatyzować. Otóż nie zawsze chcemy tworzyć parę get i set dla pola. Weźmy jako przykład pole przebieg dla naszego samochodu. Chcielibyśmy mieć możliwość pobrania aktualnego przebiegu (get), ale jego zmiana powinna być raczej efektem użycia metody jedź(liczbaKilometrów). Podobnie dla klasy Student, gdzie mielibyśmy pole średnia, którą można pobrać, ale nie ustawiamy jej ręcznie tylko liczymy na podstawie ocen. Zabezpieczy nas to przed sytuacją, kiedy ktoś ustawiłby średnią niewynikającą z posiadanych przez studenta ocen.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Ćwiczenia samodzielne

Zadanie 7.1.

Przeanalizuj samodzielnie, krok po kroku przykład klasy Car i stwórz projekt prezentujący te same mechanizmy dla klasy Student z polami name, e-mail [string], average [float/double]. Dla utrzymania porządku w projekcie proponuję stworzenie dla każdego etapu osobnej klasy w układzie:

1. Student01OneField,
2. Student02FieldsMethods,
3. Student03Constructor,
4. Student04AccessModifiers,
5. Student05Properties,
6. Student06AutoProperties.

Dzięki takiemu nazewnictwu kolejne numery pozwolą odtworzyć kolejność omawianych zagadnień, a końcówki opiszą co jest w tej konkretnej klasie nowego. Poza tym w jednej metodzie Main można przetestować wszystkie omawiane zagadnienia.

Dodatkowo można uzupełnić implementację o:

- sprawdzenie poprawności wprowadzonego adresu e-mail za pomocą wyrażeń regularnych,
- listę na oceny jedynie z get, dodawanie ocen powinno być realizowane przez osobną funkcję addNote ze sprawdzaniem czy jest to poprawna ocena. W tym wariancie średnia powinna być wyliczana na podstawie ocen bez set.

Tę implementację można zapisać w klasie

7. Student07Final.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 7.2.

Zaimplementuj klasę ułamek zwykły (Fraction).

1. Klasa ma przechowywać w polach część całkowitą, licznik i mianownik.
2. Wyświetlanie powinno uwzględniać istnienie części całkowitej (1/2 powinna się wyświetlać w formacie licznik/mianownik, a $3\frac{1}{2}$ jako całość, spacja, licznik/mianownik).
3. Ułamek musi być zawsze skrócony (nie może wyświetlać się 4/8 tylko 1/2).
4. Ułamek nie może być liczbą niewłaściwą (zamiast 8/3 powinno być 2 $\frac{2}{3}$).
5. Zaimplementuj metody add, sub, mul i div, które przyjmą jako parametr obiekt klasy ułamek i wykonają odpowiednie operacje arytmetyczne.

Zadanie 7.3.

Stwórz plik Employee.cs zawierający klasę Employee z 4 polami prywatnymi:

1. long idNumber; // numer identyfikacyjny pracownika
2. double hourlyRate; // wynagrodzenie za 1 godzinę pracy
3. double hoursWorked; // liczba przepracowanych godzin w miesiącu
4. string firstName, lastName; // imię i nazwisko pracownika

W klasie powinny znajdować się również 1 konstruktor parametrowy oraz 4 metody:

1. konstruktor ustawiający pola prywatne klasy na podstawie wartości parametrów,
2. metoda SetData ustawiająca pola prywatne klasy zgodnie z wartościami jej parametrów (sprawdź poprawność wprowadzonych danych),
3. metoda CalculateSalary obliczająca miesięczne wynagrodzenie pracownika
4. metoda Display wyświetlająca na ekranie dane osobowe pracownika oraz jego miesięczne wynagrodzenie,
5. metoda Raise zmieniająca stawkę godzinową pracownika o kwotę podaną jako parametr metody.

Konstruktor i każda metoda powinny posiadać odpowiednie komentarze.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wykorzystując konstruktor, podaj deklarację obiektu emp typu Employee, o polach zawierających dane pana Witolda Kowalskiego, mającego numer identyfikacyjny 1197, pracującego miesięcznie 170 godzin, ze stawką godzinową 30.20 zł.

Wyświetl dane pracownika.

Następnie podaj instrukcję podwyższającą stawkę godzinową pana Kota o 3.20 zł i wyświetl na ekranie dane pana Kowalskiego.

Zadanie 7.4.

Dodaj w nowym pliku EmployeeProp.cs klasę EmployeeProp. Skopiuj do niej zawartość klasy Employee i wprowadź następujące zmiany:

1. Do pól klasy użyj właściwości sprawdzając w Set poprawność danych,
2. Użyj metod Set w konstruktorze oraz w metodzie SetData,
3. Dodaj metodę ToString zamieniającą obiekt klasy EmployeeProp na napis i porównaj wyświetlanie na konsolę obiektów klasy Employee i EmployeeProp.

Zadanie 7.5.

Zdefiniuj klasę Point opisującą punkt na płaszczyźnie opisany współrzędnymi x i y. Metody tej klasy to:

1. konstruktor bezparametrowy tworzący punkt [0,0],
2. konstruktor ustawiający pola obiektu na podstawie parametrów (wartościami domyślnymi jest początek układu współrzędnych),
3. SetPoint – ustawia pola obiektu na podstawie swoich parametrów,
4. Display – wyświetla na ekranie punkt w formacie [x.xxx , y.xxx],
5. Distance() – zwraca odległość od początku układu współrzędnych
6. Distance(Point) – zwraca odległość od punktu zadanego przez parametr metody do punktu, na rzecz którego wywołana jest metoda.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 7.6.

Zdefiniuj klasę PolyLine opisującą łamaną składającą się z co najwyżej 100 punktów. Klasa zawiera statyczną tablicę punktów i pole określające liczbę punktów łamanej. Metody klasy:

1. konstruktor, którego jedynym parametrem jest liczba punktów łamanej,
2. SetSize – ustawia maksymalną liczbę punktów na podstawie swojego parametru,
3. AddPoint – dodaje jeden punkt; parametrami metody są numer punktu i jego współrzędne,
4. LoadFromFile – funkcja logiczna wczytująca punkty z pliku, którego nazwa przekazana jest przez parametr metody; jeśli nie powiodły się operacje plikowe lub nie udało się ustawienie wszystkich punktów łamanej, to wynikiem funkcji jest false, jeśli łamana została ustawiona, wynikiem jest true. Zakładamy, że w pliku znajdują się tylko liczby rzeczywiste będące współrzędnymi kolejnych punktów,
5. Display – wyświetla łamaną w formie listy punktów,
6. Length – zwraca długość łamanej.

Napisz program, w którym zdefiniowana zostanie łamana składająca się z następujących punktów:

A(1,1), B(1,-1), C(-1,-1), D(-1,1), E(1,1).

Współrzędne punktów mają zostać pobrane z pliku "data.txt". Wyświetl łamaną. Wyświetl długość łamanej.

Zadanie 7.7.

Dodaj w nowym pliku PolyLineList.cs klasę PolyLineList i zmodyfikuj klasę PolyLine tak, aby bazowała nie na tablicy, a na liście punktów.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 7.8.

Zdefiniuj klasę Matrix, która przechowuje elementy macierzy będące liczbami rzeczywistymi w tablicy jednowymiarowej. Elementy macierzy są przechowywane kolejno wierszami. Klasa powinna zawierać prywatne pola służące do przechowywania:

1. liczby wierszy,
2. liczby kolumn,
3. oraz elementów macierzy.

Ponadto w klasie powinny się znaleźć metody:

1. GetRows() – zwracająca liczbę wierszy,
2. GetColumns() – zwracająca liczbę kolumn,
3. FillRandom() – wypełniająca macierz losowymi elementami,
4. LoadFromFile(string fileName) – metoda logiczna wczytująca macierz z pliku o nazwie podanej parametrem (w pierwszej linii pliku zapisana jest liczba wierszy i kolumn, a w kolejnych liniach elementy macierzy; jeśli udało się wczytanie macierzy, metoda zwraca true),
5. SaveToFile(string fileName) – zapisująca macierz do pliku o nazwie podanej parametrem (struktura pliku jak wyżej),
6. GetElement(int row, int column) – zwracająca element macierzy znajdujący się w wierszu i kolumnie zadanych parametrami,
7. SetElement(int row, int column, double value) – ustawiająca element macierzy znajdujący się w wierszu i kolumnie zadanych parametrami,
8. konstruktor bezparametrowy (Matrix()),
9. konstruktor tworzący macierz zerową o liczbie wierszy i kolumn zadanych parametrami (Matrix(int rows, int columns)).
10. metoda ToString() zamieniającą macierz na napis.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Pytania pomocnicze

1. Jakie są zalety korzystania z programowania obiektowego.
2. Opisz filary programowania obiektowego.
3. Czym jest obiekt programistyczny?
4. Przedstaw przykładową implementację klasy owoc (Fruit)
5. Jakie znasz podstawowe modyfikatory dostępu i jak się ich używa?

Podsumowanie

Poznaliśmy pojęcie klasy jako podstawowego budulca programów, które służy do opisywania obiektów o określonych cechach i zachowaniach. Kluczowym elementem była hermetyzacja danych, realizowana poprzez używanie pól prywatnych oraz publicznych metod dostępu, co zapewnia kontrolę nad integralnością danych obiektu. Dowiedzieliśmy się, że właściwości (properties) w C# są preferowanym mechanizmem do implementacji enkapsulacji, oferując bardziej elegancką składnię niż tradycyjne metody get/set przy zachowaniu możliwości walidacji. Praktyczne zastosowanie tych koncepcji ćwiczyliśmy, tworząc przykładową klasę Car z polami takimi jak model, kolor i rok produkcji, implementując dla nich odpowiednie metody dostępu oraz konstruktor.

Literatura

1. <https://www.w3schools.com/cs/index.php>
2. https://web.mit.edu/java_v1.0.2/www/tutorial/java/objects/object.html
3. Brett D. McLaughlin, Gary Pollice, David West, *Analiza i projektowanie obiektowe. Rusz głową!*, Helion 2010



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

