PROGRAMOWANIE W JĘZYKU C#

TOMASZ WOJNAROWSKI

KLASA

Klasa konstrukcja która pozwala tworzyć, poprzez grupowanie ze sobą zmiennych właściwości i metod.

Klasa jest szablonem pozwalającym tworzyć obiekty.

KLASA

```
public class Car
{
    string make;
    int year;
    int speed;
    int maxspeed;

    public string Make { get => make; set => make = value; }
    public int Year { get => year; set => year = value; }
    public int Speed { get => speed; set => speed = value; }
    public int Maxspeed { get => maxspeed; set => maxspeed = value; }

    public Car(string make,int year,int maxspeed)...

    public void AccSpeed(int incrementSpeed)...
    public void DecSpeed(int decrementSpeed)...
}
```

WŁAŚCIWOŚCI

- Używaj właściwości, aby zapewnić prosty dostęp do prostych danych z prostymi obliczeniami.
- Nie zgłaszaj wyjątków w getterach właściwości.
- Stosuj automatycznie implementowane właściwości.
- Nie deklaruj pól publicznych

AUTOMATYCZNE WŁAŚCIWOŚCI

Kompilator umożliwia deklarację właściwości bez implementacji akcesora i deklaracji pola.

```
public int Szerokosc { get; set; }
public int Dlugosc { get; set; }
```

WŁAŚCIWOŚCI TYLKO DO ODCZYTU LUB ZAPISU

- Usunięcie gettera lub settera z właściwości pozwala zmienić sposób dostępu do niej. Właściwości z setterem pozwalają tylko na zapis. Właściwości getterem tylko na odczyt.

```
public string Get { get => Get;  }
public int Set {set => Set = value; }
```

OBIEKT

- Jest to egzemplarz klasy. Który definiują trzy podstawowe cechy:
- Tożsamość
- Stan
- Zachowanie

OBIEKT

```
Car car = new Car("TOYOTA", 2010, 155);
Console.WriteLine(car.Speed);
car.AccSpeed(100);
Console.WriteLine(car.Speed);
car.AccSpeed(100);
Console.WriteLine(car.Speed);
car.DecSpeed(100);
Console.WriteLine(car.Speed);
car.DecSpeed(100);
Console.WriteLine(car.Speed);
```

KONSTRUKTOR

- Jest to metoda uruchamiana w trakcie tworzenia obiektu.
- Inicjalizuje obiekt
- Klasa może posiadać wiele konstruktorów różniących się ilością lub typem argumentów

KONSTRUKTOR

```
public Car(string make,int year,int maxspeed)
{
    Speed = 0;
    Make = make;
    Year = year;
    Maxspeed = maxspeed;
}
public Car(string make, int year) : this(make, year, 200) { }

public Car(string make)
{
    Make = make;
}
```

DESTRUKTOR

- Jest to metoda uruchamiana w trakcie niszczenia obiektu.
- Może być tylko jeden
- Nie może być dziedziczony
- Jest wywoływany automatycznie
- Nie można stosować parametrów
- Działają we własnym wątku

```
~Car()
{
    Speed = 0;
}
```

HERMETYZACJA (ENKAPSULACJA)

- Hermetyzacja umożliwia ukrywanie szczegółów. W razie konieczności można uzyskać do nich dostęp, jednak dzięki inteligentnemu ukryciu szczegółów zrozumienie dużych programów jest łatwiejsze, dane są chronione przed przypadkową modyfikacją, a kod staje się prostszy w konserwacji, ponieważ skutki modyfikacji kodu są ograniczone do hermetycznej jednostki.

HERMETYZACJA (ENKAPSULACJA)

```
string make;
int year;
int speed;
int maxspeed;

public string Make { get => make; set => make = value; }
public int Year { get => year; set => year = value; }
public int Speed { get => speed; set => speed = value; }
public int Maxspeed { get => maxspeed; set => maxspeed = value; }
```

DZIEDZICZENIE

- Mechanizm współdzielenia funkcjonalności. Klasa która dziedziczy po innej klasie, że oprócz swoich własnych atrybutów i właściwości i metod, uzyskuje także te pochodzące z klasy, z której bazowej.

DZIEDZICZENIE

```
public class Vehicle
{
    string make;
    int year;
    int speed;
    int maxspeed;

    public string Make { get => make; set => make = value; }
    public int Year { get => year; set => year = value; }
    public int Speed { get => speed; set => speed = value; }
    public int Maxspeed { get => maxspeed; set => maxspeed = value; }
}
```

```
public class Car:Vehicle
{
    public Car(string make,int year,int maxspeed)...
    public Car(string make, int year) : this(make, year, 200) { }
    public Car(string make)...
    public void AccSpeed(int incrementSpeed)...
    public void DecSpeed(int decrementSpeed)...

    ~Car()...
}
```

OPERATOR this

- Pozwala odwołać się do bieżącej instancji klasy. Stosowanie poprawia czytelność kodu

```
public Car(string Make,int Year,int Maxspeed)
{
    Speed = 0;
    this.Make = Make;
    this.Year = Year;
    this.Maxspeed = Maxspeed;
}
```

MODYFIKATOR const

- Pozwala deklarować zmienną niemodyfikowalną
- Nie może być stosowana z modyfikatorem static
- Przypisywana w czasie kompilacji
- Wartość inicjalizowana w deklaracji

MODYFIKATOR readonly

- Pozwala deklarować zmienną niemodyfikowalną
- Może być stosowana z modyfikatorem static
- Inicjalizowana w czasie wykonywania
- Wartość inicjalizowana w deklaracji lub konstruktorze

MODYFIKATORY DOSTĘPU

Modyfikator	Opis
public	Publiczne składniki klasy są dostępne dla wszystkich metod wszystkich klas
private	Składowe prywatne są dostępne tylko wewnątrz klasy
protected	Składowe dostępne w klasie bazowej i we wszystkich jej klasach pochodnych
internal	Składowe dostępne wewnątrz tej samej biblioteki

KLASY STATYCZNE

- Klasa statyczna musi składać się metod i pól statycznych.
- Klasa może posiadać jeden konstruktor statyczny
- Nie można utworzyć instancji klasy statycznej
- W deklaracji używamy słowa static

KONSTRUKTOR STATYCZNY

- Może być zdefiniowany tylko jeden
- Nie może posiadać parametrów
- Nie ma żadnego modyfikatora dostępu
- Wykonuje się przed zwykłymi konstruktorami.

INTERFEJS

- Deklarowane za pomocą słowa ‚kluczowego interface
- Nie mogą zawierać implementacji
- Zawierają zestaw składowych określających możliwości klasy implementującej,
- Interfejsy mogą dziedziczyć po wielu interfejsach.

INTERFEJS

```
interface IVehicle
{
    void Start();
    void Stop();
    void AccSpeed(int incrementSpeed);
    void DecSpeed(int decrementSpeed);
}
```

KLASY ABSTRAKCYJNE

- Deklarowane za pomocą słowa kluczowego abstract dodanego do definicji klasy
- Składają się ze zmiennych abstrakcyjnych;
- Nie można utworzyć instancji klasy bazowej
- Interfejsy mogą dziedziczyć po wielu interfejsach.

KLASY ABSTRAKCYJNE

```
public abstract class Vehicle
{
    protected string make;
    protected int year;
    protected int speed;
    protected int maxspeed;
    private bool engineOn;
    public virtual string Make { get => make; set => make = value; }
    public virtual int Year { get => year; set => year = value; }
    public virtual int Speed { get => speed; set => speed = value; }
    public virtual int Maxspeed { get => maxspeed; set => maxspeed = value; }
    protected bool EngineOn { get => engineOn; set => engineOn = value; }

    public virtual void PrintSpeed()
    {
        Console.WriteLine("Aktualna prędkość pojazdu wynosi {0}", Speed);
    }
    public abstract void Start();
    public abstract void Stop();
    public abstract void AccSpeed(int incrementSpeed);
    public abstract void DecSpeed(int decrementSpeed);
}
```

METODY WIRTUALNE

- Deklarowane za pomocą słowa kluczowego virtual
- Aby przesłonić metodę virtualną należy użyć słowa ovveride w klasie pochodnej
- Nie ma obowiązku przesłaniania metod wirtualnych
- Należy unikać metod wirtualnych dla sekcji krytycznych w programach.

METODY WIRTUALNE

```
public virtual void PrintSpeed()
{
    Console.WriteLine("Aktualna prędkość pojazdu wynosi {0}", Speed);
}
```

```
public override void PrintSpeed()
{
    base.PrintSpeed();
    if (Speed == 0)
        Console.WriteLine("Stoisz w miejscu");
    else if (Speed > 90)
        Console.WriteLine("Jedziesz za szybko! Zwolnij");

}
```

OPERATOR is

- Operator is pozwala sprawdzić czy dane są określonego typu.

```
private static bool IsVehicle(object data)
{
    if (data is Vehicle)
    {
        Console.WriteLine(" jest pojazdem!");
        return true;
    }
    else
    {
        Console.WriteLine(" nie jest pojazdem!");
        return false;
    }
}
```

OPERATOR as

- Operator próbuje przeprowadzić konwersję na określony
- typ i ustawia wartość null, jeśli źródłowej wartości nie przekształcić na docelowy typ..

```
private static void PrintSpeed(object data)
{
    Vehicle vehicle = data as Vehicle;
    if (vehicle is null)
        Console.WriteLine(" nie jest pojazdem! Nie możemy poinformować Ciebie o jego prędkości.");
    else
        vehicle.PrintSpeed();
}
```

METODY ROZSZERZAJĄCE

- Metody pozwalają na dodanie funkcjonalności do istniejących typow bez dziedziczenia
- Są statyczne
- Muszą być zadeklarowana w klasie statycznej
- Pierwszym parametrem jest operator this, który informuje że jest to metoda rozszerzająca
- Metody rozszerzające nie przesłaniają istniejących metod

Struktury (struct)

- Inicjalizalowane słowem kluczowym struct.
- Mogą obejmować pola, właściwości, metody i konstruktory.
- Automatyczne implementowanie właściwości pól jako tylko do odczytu
- Należy unikać modyfikowania typów bezpośrednich.
- Nie można inicjować zmiennych w deklaracji.

DELEGATY

- Deklarowane za pomocą słowa kluczowego delegate
- Delegaty przechowują referencję do metody

```
delegate void Print();
static void Main(string[] args)
{
    Car car = new Car("Ford",2016,210);
    Print print = car.PrintMake;
    print += car.PrintYear;
    print += car.PrintSpeed;
    print();
    print -= car.PrintYear;
    print();
}
```

KOLEKCJE

- Zapewniają elastyczny sposób pracy z grupami obiektów
- Można zwiększać i zmniejszać ich rozmiar dynamicznie.
- Należy zadeklarować kolekcję przed dodaniem do niej elementów.
- Znajdują się w przestrzeni nazw System.Collections

KOLEKCJE

Klasa	Opis
Queue	Reprezentuje kolekcję obiektów(kolejkę) FIFO
Stack	Reprezentuje kolekcję obiektów(stos) LIFO
ArrayList	Implementuje IList za pomocą tablicy, której rozmiar jest zwiększany dynamicznie.
SortedList	Reprezentuje kolekcję pary klucz wartość, które są posortowane według kluczy i są dostępne przez klucz i indeksu.
HashTable	Reprezentuje kolekcję pary klucz wartość, które są podzielone na podstawie kodu skrótu klucza.

TYPY GENERYCZNE

- Zapewniają mechanizm tworzenia struktur danych, które można przekształcić na wyspecjalizowaną wersję w celu obsługi konkretnych typów
- Umożliwiają jednokrotną implementację algorytmów i wzorców
- Parametr określający typ (T) należy podać w nawiasie ostrym po nazwie klasy

TYPY GENERYCZNE

```
class GenShow<T>
{
    public void Show(T variable)
    {
        Console.WriteLine(" typ: {0}", variable.GetType());
        Console.WriteLine(" wartość: {0}", variable);
    }
}
```

```
static void Main(string[] args)
{
    GenShow<double> myGeneric = new GenShow<double>();
    myGeneric.Show(1.12);
    GenShow<int> myGenericInt = new GenShow<int>();
    myGenericInt.Show(64);

}
```

TYPY GENERYCZNE – ZALETY STOSOWANIA

- Typy generyczne zwiększają bezpieczeństwo (typ jest ściśle określone)
- Sprawdzanie typu następuje na etapie kompilacji
- Typy bezpośrednie nie są opakowywane do typu object
- Zwiększenie wydajności (brak opakowywania)
- Poprawiają czytelność kodu
- Zmniejszają ilość pamięci

TYPY GENERYCZNE – QUEUE

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("Pierwszy");
queue.Enqueue("Drugi");
queue.Enqueue("Trzeci");
Console.WriteLine(queue.FirstOrDefault());
foreach (string text in queue)
    Console.WriteLine(text);
```

TYPY GENERYCZNE – STACK

```
Stack<string> stack = new Stack<string>();
stack.Push("Pierwszy");
stack.Push("Drugi");
stack.Push("Trzeci");
Console.WriteLine(stack.FirstOrDefault());
foreach (string text in stack)
    Console.WriteLine(text);
```

TYPY GENERYCZNE – LIST

```
static void Main(string[] args)
{
    List<string> list = new List<string>();
    list.Add("Pierwszy");
    list.Add("Drugi");
    list.Add("Trzeci");
    list.Add("Czwarty");
    Console.WriteLine(list.FirstOrDefault());
    Console.WriteLine(list.FirstOrDefault());
    list.Remove("Trzeci");
    list.Sort();
    foreach (string text in list)
        Console.WriteLine(text);
}
```

TYPY GENERYCZNE – LIST

```
static void Main(string[] args)
{
    List<string> list = new List<string>();
    list.Add("Pierwszy");
    list.Add("Drugi");
    list.Add("Trzeci");
    list.Add("Czwarty");
    Console.WriteLine(list.FirstOrDefault());
    Console.WriteLine(list.FirstOrDefault());
    list.Remove("Trzeci");
    list.Sort();
    foreach (string text in list)
        Console.WriteLine(text);
}
```