Politechnika Wrocławska

Programowanie obiektowe

Wykład 9

Klasy abstrakcyjne

Interfejsy

Modyfikatory dostępu

Prowadzący
dr inż. Paweł Maślak

Plan wykładu

- Klasy abstrakcyjne
- Interfejsy
- Modyfikatory dostępu
- Właściwości i akcesory *get* oraz *set*

Klasy abstrakcyjne

- Klasy abstrakcyjne są jednym z podstawowych elementów programowania obiektowego w języku C#. Są to klasy, które nie mogą być bezpośrednio instancjonowane, czyli nie można utworzyć obiektu klasy abstrakcyjnej. Ich głównym celem jest umożliwienie programistom definiowania szkieletów klas, które wymagają implementacji przez klasy pochodne.

Klasy abstrakcyjne

- W praktyce oznacza to, że abstrakcyjna klasa definiuje zestaw metod i właściwości, które muszą być zdefiniowane w każdej klasie dziedziczącej po niej. Dzięki temu programista może uniknąć powtarzalnego kodu i zapewnić spójność w całej hierarchii dziedziczenia. Aby zdefiniować klasę abstrakcyjną w C#, należy użyć słowa kluczowego abstract przed deklaracją klasy.

Klasy abstrakcyjne

- Klasa abstrakcyjna Vehicle, zawierająca dwie metody abstrakcyjne – Start() oraz Stop() i jedną metodę wirtualną – Drive()

```
public abstract class Vehicle
{
    public abstract void Start ();
    public abstract void Stop();
    public virtual void Drive()
    {
        Console.WriteLine("Vehicle is driving");
    }
}
```

Klasy abstrakcyjne

- Klasa abstrakcyjna Vehicle, zawierająca dwie metody abstrakcyjne – Start() oraz Stop() i jedną metodę wirtualną – Drive()

```
public abstract class Vehicle
{
    public abstract void Start ();
    public abstract void Stop();
    public virtual void Drive()
    {
        Console.WriteLine("Vehicle is driving");
    }
}
```

Klasy abstrakcyjne

- Pierwsze dwie metody są abstrakcyjne, co oznacza, że muszą zostać zaimplementowane w klasach pochodnych.
- Trzecia metoda jest wirtualna, co oznacza, że może być przesłonięta przez klasy dziedziczące, ale nie musi. Klasy dziedziczące po klasie abstrakcyjnej muszą zaimplementować wszystkie abstrakcyjne metody i właściwości zdefiniowane przez klasę abstrakcyjną.

Klasy abstrakcyjne

- Implementacja metod abstrakcyjnych Start() oraz Stop() w klasie pochodnej Car, dziedziczącej klasę bazową Vehicle

```
public class Car : Vehicle
{
    public override void Start()
    {
        Console.WriteLine("Car is starting");
    }

    public override void Stop()
    {
        Console.WriteLine("Car is stopping");
    }
}
```

Klasy abstrakcyjne

- W powyższym przykładzie klasa Car dziedziczy po klasie abstrakcyjnej Vehicle i implementuje jej abstrakcyjne metody Start() i Stop(). Klasy abstrakcyjne mogą także zawierać konstruktory, ale nie mogą być wywoływane bezpośrednio, ponieważ są klasy abstrakcyjne.

Klasy abstrakcyjne

- Pozwalają na tworzenie szablonów dla klas, które wymagają wspólnej funkcjonalności. Na przykład, jeśli tworzymy grę z wieloma rodzajami pojazdów, możemy stworzyć klasę abstrakcyjną Vehicle, która zawiera metody takie jak Start(), Stop() i Drive(), a następnie dziedziczyć po niej klasy takie jak Car, Bike czy Plane, które implementują konkretne funkcjonalności dla każdego rodzaju pojazdu.

Klasy abstrakcyjne

- Użycie metody klasy bazowej, na obiekcie klasy pochodnej

```
Vehicle car = new Car();
car.Start();
```

- W powyższym przykładzie utworzony został obiekt klasy Car, która dziedziczy po klasie abstrakcyjnej Vehicle, obiekt ten – car – został utworzony jako typu Vehicle, ale zainicjowany jako obiekt typu Car.
- W ten sposób można użyć metody Start zdefiniowanej w klasie abstrakcyjnej Vehicle dla obiektu car.

Klasy abstrakcyjne

- W aplikacjach biznesowych klasa abstrakcyjna może być używana do definiowania interfejsów dla różnych rodzajów baz danych lub integracji z systemami zewnętrznymi.
- W grach klasa abstrakcyjna może definiować wspólną funkcjonalność dla różnych rodzajów postaci lub obiektów. Klasy abstrakcyjne to ważne narzędzie w programowaniu obiektowym w C#. Dzięki nim programista może uniknąć powtarzalnego kodu i zapewnić spójność w całej hierarchii dziedziczenia.

Interfejsy

- Interfejsy w języku C# służą do definiowania kontraktów, które implementują klasy. Definiują one zestaw metod i właściwości, które klasa musi zaimplementować, aby spełnić kontrakt interfejsu. Interfejsy pozwalają na tworzenie kodu bardziej elastycznego i łatwiejszego do utrzymania, ponieważ klasa może implementować wiele interfejsów jednocześnie.

Interfejsy

- Interfejs **IAnimal**

```
public interface IAnimal
{
    string Name;
    void Move();
}
```

- W interfejsie **IAnimal** definiujemy dwa elementy: właściwość **Name** typu **string** oraz metodę **Move**(), która nie zwraca wartości.

Interfejsy

- Implementacja interfejsu **IAnimal** przez klasę **Dog**

```
public class Dog : IAnimal
{
   public string Name;

   public void Move()
   {
      Console.WriteLine("The dog is running.");
   }
}
```

- W powyższym przykładzie klasa Dog implementuje interfejs IAnimal. Definiujemy w tej klasie właściwość Name oraz metodę Move(), które są wymagane przez interfejs IAnimal.

Interfejsy

Użycie obiektu klasy **Dog** jako obiektu typu **Ianimal**

```
IAnimal myAnimal = new Dog();
myAnimal.Name = "Burek";
myAnimal.Move();
```

- Dzięki temu, że klasa Dog implementuje interfejs IAnimal, możemy traktować obiekt typu Dog jako obiekt typu IAnimal. W tym przypadku tworzymy obiekt Dog, ale przypisujemy go do zmiennej typu IAnimal. Następnie ustawiamy wartość właściwości Name i wywołujemy metodę Move(), która jest zdefiniowana w interfejsie IAnimal.

Modyfikatory dostępu

- Modyfikatory dostępu to kluczowe narzędzia w języku C#, które pozwalają programistom kontrolować, jakie elementy klasy i metod są dostępne w innych częściach programu. Dzięki modyfikatorom dostępu, programista może zdecydować, czy dana klasa lub metoda jest publiczna i dostępna z każdego miejsca w programie, czy prywatna i dostępna tylko w obrębie klasy.

Modyfikatory dostępu

- Modyfikatory te pozwalają również na kontrolowanie dziedziczenia, zapewniając, że pewne elementy klasy są dziedziczone tylko przez jej potomków. W języku C# występują cztery modyfikatory dostępu:
    1) public,
    2) private,
    3) protected
    4) internal.
- Każdy z nich ma swoje unikalne właściwości i zastosowania. Stosowanie odpowiedniego modyfikatora dostępu jest ważne dla zapewnienia bezpieczeństwa i spójności kodu oraz ułatwia jego czytanie i utrzymywanie.

Modyfikatory dostępu - Public

- Modyfikator public oznacza, że dana klasa lub metoda jest dostępna z każdego miejsca w programie. Oznacza to, że kod innych klas lub metod może korzystać z publicznych elementów klasy, a także tworzyć obiekty klasy i wywoływać jej publiczne metody. Public jest najbardziej otwartym i ogólnym modyfikatorem dostępu i często stosuje się go w przypadku klas i metod, które są kluczowe dla funkcjonalności programu

Modyfikatory dostępu - Public

- Przykład użycia modyfikatora public:

```
public class ExampleClass
{
    public void PublicMethod()
    {
        Console.WriteLine("Public method");
    }
}
```

- W tym przykładzie, klasa ExampleClass zawiera jedną publiczną metodę PublicMethod(), która może być wywołana z innych części programu. W języku programowania C#, modyfikator dostępu public jest jednym z najważniejszych elementów programowania obiektowego, jest najbardziej ogólnym modyfikatorem dostępu, co oznacza, że składowe klasy oznaczone jako public są dostępne z każdego miejsca w programie.

Modyfikatory dostępu - Public

- Klasy, które zawierają składowe oznaczone jako public, mogą być wykorzystywane przez inne klasy i obiekty. Dzięki temu można tworzyć hierarchie klas, w których każda kolejna klasa wykorzystuje klasy i składowe z poprzednich klas. Używanie modyfikatora public może nie być zawsze bezpieczne i należy pamiętać o zasadzie najmniejszego uprawnienia.
- Oznaczając składowe klasy jako public, umożliwiamy dostęp do nich z każdego miejsca w programie, co może prowadzić do nieoczekiwanych błędów.

Modyfikatory dostępu - Private

- Modyfikator private oznacza, że dana klasa lub metoda jest dostępna tylko w obrębie klasy, w której została zdefiniowana. Oznacza to, że kod innych klas lub metod nie może korzystać z prywatnych elementów klasy ani tworzyć obiektów klasy. Modyfikator **private** jest najbardziej ograniczającym modyfikatorem dostępu i często stosuje się go w przypadku metod pomocniczych lub prywatnych pól.

Modyfikatory dostępu - Private

- Przykład użycia modyfikatora private:

```
public class ExampleClass
{
    private void PrivateMethod()
    {
        Console.WriteLine("Private method");
    }
}
```

- W przykładzie tym, klasa ExampleClass zawiera jedną prywatną metodę PrivateMethod(), która jest dostępna tylko w obrębie klasy. Warto zaznaczyć, że modyfikator private zapewnia nie tylko bezpieczeństwo danych, ale również umożliwia łatwiejsze utrzymanie kodu.

Modyfikatory dostępu - Private

- Używając modyfikatora private, można stworzyć interfejs klasy, który jest bardziej przejrzysty i czytelny dla innych programistów. Jednakże, warto zauważyć, że zastosowanie modyfikatora private może prowadzić do problemów w przypadku dziedziczenia klas. W przypadku, gdy dziedzicząca klasa potrzebuje dostępu do pól lub metod klasy bazowej oznaczonych jako private, konieczne jest wykorzystanie innego modyfikatora.

Modyfikatory dostępu - Protected

- Modyfikator **protected** oznacza, że dana metoda lub pole jest dostępne tylko w obrębie klasy, w której zostało zdefiniowane oraz w klasach potomnych. Oznacza to, że kod innych klas lub metod nie może korzystać z chronionych elementów klasy ani tworzyć obiektów klasy. Modyfikator protected jest często stosowany w przypadku dziedziczenia, gdy metody lub pola muszą być widoczne tylko dla klas potomnych.

Modyfikatory dostępu - Protected

- Przykład użycia modyfikatora protected

```
public class ExampleClass
{
    protected void Protected()
    {
        Console.WriteLine("Protected method");
    }
}
public class ChildClass : ExampleClass
{
    public void ChildMethod()
    {
        Protected(); //metoda chroniona jest dostępna w klasie potomnej
    }
}
```

Modyfikatory dostępu - Protected

- W przykładzie tym, klasa ExampleClass zawiera jedną prywatną metodę PrivateMethod(), która jest dostępna tylko w obrębie klasy. Warto zaznaczyć, że modyfikator private zapewnia nie tylko bezpieczeństwo danych, ale również umożliwia łatwiejsze utrzymanie kodu.

Modyfikatory dostępu - Protected

- Modyfikator protected umożliwia dostęp do składowych klasy z poziomu klasy dziedziczącej, ale jednocześnie zapewnia kontrolę nad tym, które składowe klasy są udostępnione klasom dziedziczącym, a które są chronione przed zmianą przez te klasy.

Modyfikatory dostępu - Protected

- Modyfikator protected znajduje zastosowanie w przypadku, gdy chcemy stworzyć hierarchię dziedziczenia, w której klasy dziedziczące mają dostęp do niektórych składowych klasy bazowej. Przykładowo, jeśli mamy klasę Animal, to jej składowe, takie jak age oraz species, mogą być oznaczone jako protected, aby klasy dziedziczące, takie jak na przykład Dog, miały dostęp do tych składowych i mogły je zmieniać.

Modyfikatory dostępu - Protected

- Przykład użycia modyfikatora protected – metoda Move() jest oznaczona właśnie tym modyfikatorem dostępu

```
public class Animal
{
    protected int age;
    protected string species;
    protected void Move()
    {
        // kod metody
    }

    protected Animal()
    {
        // kod konstruktora
    }
}
```

```
public class Dog : Animal
{
    public void Bark()
    {
        // kod metody
    }

    public Dog() : base()
    {
        // kod konstruktora
    }
}
```

Modyfikatory dostępu - Protected

- W tym przykładzie składowe klasy Animal oznaczone są jako protected, co oznacza, że klasy dziedziczące, takie jak Dog, mogą mieć do nich dostęp. Klasa Dog dziedziczy również konstruktor klasy Animal oraz metodę Move(), która jest chroniona i dostępna tylko z poziomu klasy dziedziczącej.

Modyfikatory dostępu - Internal

- Modyfikator internal oznacza, że dana klasa lub metoda jest dostępna tylko w obrębie bieżącej biblioteki. Oznacza to, że kod innych bibliotek nie może korzystać z wewnętrznych elementów klasy ani tworzyć obiektów klasy.
- Modyfikator ten, jest często stosowany w przypadku klas i metod, które są wewnętrzne dla biblioteki i nie powinny być używane poza nią.

Modyfikatory dostępu - Internal

- Przykład użycia modyfikatora internal:

```
internal class ExampleClass
{
    internal void InternalMethod()
    {
        Console.WriteLine("Internal method");
    }
}
```

- W tym przykładzie, klasa ExampleClass jest oznaczona jako wewnętrzna i zawiera wewnętrzną metodę InternalMethod(), która jest dostępna tylko w obrębie tej samej biblioteki. Użycie modyfikatora internal jest szczególnie przydatne w dużych projektach, gdzie chcemy kontrolować dostęp do poszczególnych elementów zdefiniowanych w danym projekcie. Dzięki temu można ukryć pewne elementy przed innymi projektami i zapobiec sytuacjom, w których elementy te są nieumyślnie modyfikowane lub wykorzystywane w sposób nieprawidłowy.

Modyfikatory dostępu - Internal

- W praktyce, użycie internal pozwala na uproszczenie kodu poprzez ukrycie niepotrzebnych elementów przed innymi projektami. Modyfikator internal może być stosowany do pól klasy, właściwości, metod i klas.

Modyfikatory dostępu - Internal

- Przykład użycia modyfikatora internal – klasa MyClass dostępna jest tylko wewnątrz projektu, w którym ta klasa się znajduje

```
internal class MyClass
{
    internal int myField;
    internal void MyMethod()
    {       // kod metody
    }
}
public class Program
{
    static void Main(string[] args)
    {
        MyClass myObj = new MyClass();
        myObj.myField = 10; // dostępny tylko w obrębie projektu
        myObj.MyMethod();   // dostępny tylko w obrębie projektu
    }
}
```

Właściwości i akcesory *get/set*

- Właściwość to konstrukcja charakterystyczna dla języka C#. Zapewnia dostęp do pól klasy posługując się przy tym akcesorami **get** i **set**. Główną funkcjonalnością właściwości jest możliwość zapisywania i odczytywania prywatnych pól klasy, tak jak by były publiczne.
- Jednym z podstawowych filarów programowania obiektowego jest **hermetyzacja**, która mówi, aby ukrywać składniki klasy, po to by nie były dostępne z poziomu innej klasy.

Właściwości i akcesory *get/set*

- Przykład metod, których celem jest pobranie wartości zmiennej _age oraz ustawienie tej wartości

```
class Person
{
    private int _age;

    public int GetPersonAge()
    {
        return _age;
    }
    public void SetPersonAge(int age)
    {
        _age = age;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Person person = new Person();

        person.SetPersonAge(21);

Console.WriteLine(person.GetPerson
Age())
            ;
    }
}
```

Właściwości i akcesory *get/set*

```
class Person
{
   private int _age;

   public int GetPersonAge()
   {
      return _age;
   }
   public void SetPersonAge(int age)
   {
      _age = age;
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      Person person = new Person();

      person.SetPersonAge(21);

Console.WriteLine(person.GetPerson
Age())
         ;
   }
}
```

```
21

Process finished with exit code 0
```

- Można zauważyć, że w powyższym przykładzie, trzymamy się zasad poprawnego programowania obiektowego. Przykład jest dość prosty. Pole _age jest zmienną prywatną i żeby je zmodyfikować musimy posłużyć się tylko metodami, które udostępnia klasa, nazywamy je „getterami” i „setterami”.

Właściwości i akcesory *get/set*

- Zmienna prywatna oraz przypisane do niej metody do odczytu i zapisu dają pewność, że inny programista pracujący na naszej klasie, będzie trzymał się naszych założeń.

Właściwości i akcesory *get/set*

- Możemy rozbudować metodę, aby akceptowalny wiek był z przedziału od 0 do 100:

```
class Person
{
   private int _age;

   public int GetPersonAge()
   {
      return _age;
   }
   public void SetPersonAge(int age)
   {
if (age => 0 && age <= 100)
     {
        _age = age;
     }
   }
}
```

Właściwości i akcesory *get/set*

- Właściwość w języku C# deklarujemy w sposób podobny do zmiennej, jednak w jej wnętrzu należy obsłużyć akcesory get oraz set. Używając właściwości, nie będzie potrzeby pisać samemu poszczególnych metod do każdego pola.

Właściwości i akcesory *get/set*

- Klasa Person używająca właściwości:

```
using System;

class Person
{
   // właściwość
   public int age
   {
      get
      {
         return age;
      }
      set
      {
         age = value;
      }
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      Person person = new Person();
      person.age = 21;
      Console.WriteLine(person.age);
   }
}
```

Właściwości i akcesory *get/set*

- Oba omówione przykłady działają tak samo. W pierwszym użyto zwykłych metod a w drugim właściwości. W kodzie pojawiły się akcesory get oraz set. Pole age nadal jest prywatne, ale właściwość age jest publiczna i zapewnia ona dostęp do prywatnego pola _age.
- Ważne aby pamiętać, że akcesor get wywoływany jest w chwili, gdy chcemy pobrać wartość właściwości, a akcesor set wywołany jest w chwili nadania wartości tejże właściwości.

Właściwości i akcesory *get/set*

- Warto zwrócić uwagę na słowo kluczowe value przy akcesorze set, ma ono szczególną funkcję jedynie wewnątrz ciała właściwości - reprezentuje wartość przypisywaną do właściwości w akcesorze set. Od tej chwili programista komunikuje się z polami klasy za pomocą publicznych właściwości, mimo tego kod nadal jest stosunkowo długi, ale na szczęście wprowadzona została deklaracja skrócona. Pozwala ona zredukować ilość kodu do absolutnego minimum w prostych programach.

Właściwości i akcesory *get/set*

- Klasa Person używająca właściwości w formie skróconej:

```
class Person
{
   // właściwość
   public int age { get; set; }

   class Program
   {
      static void Main(string[] args)
      {
         Person person = new Person();
         person.age = 21;
         Console.WriteLine(person.age);
      }
   }
```

Właściwości i akcesory *get/set*

- Środowisko developerskie automatycznie rozwinie skróconą deklarację właściwości do normalnej postaci podczas kompilacji. Kompilator niejawnie utworzy nawet pole prywatne przypisane do publicznej właściwości.
- Powyższy program działa identycznie tak jak w poprzednich przykładach. Skrócona forma właściwości, jest bardziej przydatna, na przykład tworząc klasę Person o właściwościach name, surname i age w klasie posiadamy tylko trzy linijki kodu. Trzeba pamiętać jednak, że we właściwościach nie można, a przynajmniej nie powinno się umieszczać zaawansowanych fragmentów kodu – dozwolone są tylko co najwyżej proste operacje.

Politechnika Wrocławska

Dziękuje bardzo za uwagę

Dr inż. Paweł Maślak