Politechnika Wrocławska

Programowanie obiektowe

Wykład 7

Programowanie obiektowe

Konstruktory Destruktory

Prowadzący
dr inż. Paweł Maślak

Plan wykładu

- Programowanie obiektowe
- Konstruktor
- Destruktor
- Tworzenie obiektów

Programowanie obiektowe

Programowanie obiektowe to jedna z najważniejszych koncepcji w dziedzinie informatyki, a w języku C# jest ono jednym z podstawowych stylów programowania. Programowanie obiektowe (OOP – z ang. Object Oriented Programming) to styl programowania, który koncentruje się na tworzeniu obiektów – polega na tworzeniu klas, które zawierają pola i metody.

Pola Metody

- Pola przechowują dane, a metody definiują zachowanie klasy.
- W programowaniu obiektowym korzysta się z klasy jako szablonu, aby utworzyć obiekt.
- Obiekt jest instancją klasy. Kiedy zostanie utworzony obiekt klasy, będzie on zawierał wszystkie pola i metody, które są zdefiniowane w klasie. Można użyć tych metod, aby wykonać różne czynności na obiekcie lub zmienić jego stan.

Klasa

- Klasa to szablon, który definiuje zestaw pól i metod. Kiedy utworzy się obiekt z klasy, będzie on miał te same pola i metody, które są zdefiniowane w klasie. Klasa może mieć pola, które przechowują dane, oraz metody, które wykonują różne czynności na tych danych.
- Klasa może być również zagnieżdżona w innej klasie lub pliku, co pozwala na bardziej czytelny i modułowy kod.

Dziedziczenie

- Dziedziczenie to proces, w którym jedna klasa dziedziczy pola i metody z innej klasy. Klasa, która dziedziczy, nazywana jest podklasą, a klasa, z której dziedziczy, nazywana jest nadklasą lub klasą bazową.
- Dziedziczenie w C# pozwala na tworzenie bardziej specjalizowanych klas na podstawie istniejących klas. Podklasy dziedziczą pola i metody z nadklasy, co umożliwia programistom ponowne wykorzystanie kodu.

Polimorfizm

- Polimorfizm to zdolność do wykorzystywania jednej nazwy metody do realizacji różnych zadań. W C# polimorfizm pozwala na przesłanianie metod, czyli definiowanie nowej implementacji metody w podklasie, która zastępuje implementację metody w nadklasie.
- Polimorfizm w C# pozwala na bardziej elastyczne i rozszerzalne kodowanie i definiowanie zachowania obiektów w zależności od kontekstu.

Enkapsulacja

- Enkapsulacja to proces ukrywania szczegółów implementacyjnych klasy i udostępniania tylko niezbędnych interfejsów, które umożliwiają korzystanie z tych szczegółów.
- W C# enkapsulacja osiąga się poprzez użycie modyfikatorów dostępu.

Abstrakcja

- Abstrakcja to proces wyodrębniania istotnych cech obiektów i ignorowania reszty. W C# abstrakcje osiąga się poprzez użycie interfejsów i klas abstrakcyjnych.
- Abstrakcja w C# pozwala na tworzenie bardziej ogólnych klas i interfejsów, które są łatwiejsze do utrzymania i rozwijania oraz koncentrację na istotnych cechach obiektów, a ignorowanie reszty.

Abstrakcja

- Abstrakcja to proces wyodrębniania istotnych cech obiektów i ignorowania reszty. W C# abstrakcje osiąga się poprzez użycie interfejsów i klas abstrakcyjnych.
- Abstrakcja w C# pozwala na tworzenie bardziej ogólnych klas i interfejsów, które są łatwiejsze do utrzymania i rozwijania oraz koncentrację na istotnych cechach obiektów, a ignorowanie reszty.

Klasa

- Klasa jest tworzona tylko raz, a obiekty są tworzone wielokrotnie w oparciu o tę samą definicję. Klasa pozwala na tworzenie obiektów z określonymi właściwościami i konkretnym zachowaniem.
- Na przykład, jeśli chcemy stworzyć program, który będzie zarządzał listą studentów, możemy stworzyć klasę Student, definiującą właściwości i zachowania typowe dla studenta.

Klasa - cd

- Właściwości te mogą obejmować imię, nazwisko, numer indeksu i datę urodzenia.
- Zachowania zaś mogą obejmować dodawanie, usuwanie i aktualizowanie studentów na liście oraz obliczanie średniej ocen.
- Definiując klasę, można również określić relacje między różnymi klasami.

Klasa - cd

- Przykład: program musi zarządzać listą przedmiotów, możemy stworzyć klasę Subject, a następnie dodać relację wiele do wielu między klasami Student i Subject, co pozwoli na gromadzenie informacji o tym, jakie przedmioty są dla danego studenta.

Klasa - cechy

- abstrakcyjne i modularne projektowanie programów oraz definiowanie typów danych i zachowań
- hermetyzacja kodu
- ułatwia testowanie kodu
- łatwe grupowanie podobnych funkcjonalności i oddzielanie ich od innych części kodu

Klasa

- Klasa w programowaniu obiektowym służy do definiowania szablonu lub wzorca, na podstawie którego tworzone są **obiekty**.
- Jest to szkielet, który określa, co składa się na obiekt oraz jakie operacje można na nim wykonywać.
- W skład klasy wchodzą **metody** i **zmienne**, które nazywamy **składowymi**.

Klasa

- Definicja klasy zaczyna się od słowa kluczowego class, po którym podajemy nazwę klasy.
- Wewnątrz klasy umieszczamy definicje **pól**, czyli zmiennych, które będą przechowywać dane charakterystyczne dla obiektów tej klasy oraz definicje **metod**, które będą operować na tych danych.

Klasa - przykład

- Przykładowo, dla klasy Person, mogą to być pola **name** i **age**, przechowujące odpowiednio imię i wiek osoby, oraz metody, takie jak **getName**() i **getAge**(), które zwracają wartości tych pól.
- **Definicja klasy** zaczyna się od słowa kluczowego **class**, po którym następuje nazwa tejże klasy. Wnętrze klasy jest otoczone parą nawiasów klamrowych.

Klasa - schemat

```
modyfikator_dostepu class nazwa_klasy
{
 // pola klasy
 modyfikator_dostepu typ_danych zmienna1;
 modyfikator_dostepu typ_danych zmienna2;
 ...
 modyfikator_dostepu typ_danych zmiennaN;
 // metody klasy
 modyfikator_dostepu zwracany_typ metoda1(lista_parametrow)
 {
   wnetrze_metody
 }
 modyfikator_dostepu zwracany_typ metoda2(lista_parametrow)
 {
   wnetrze_metody
 }
 ...
 modyfikator_dostepu zwracany_typ metodaN(lista_parametrow)
 {
   wnetrze_metody
 }
}
```

Konstruktor - Destruktor

- W języku programowania C#, konstruktor i destruktor są dwoma podstawowymi elementami programowania obiektowego.
- Konstruktor służy do inicjalizacji obiektów
- Destruktor jest używany do zwalniania zasobów, gdy obiekt jest usuwany z pamięci

Konstruktor

- **Konstruktor** jest specjalną metodą klasy, która jest wywoływana, gdy tworzony jest nowy obiekt danej klasy. Jest on używany do inicjalizacji stanu obiektu, takiego jak ustawianie wartości domyślnych dla zmiennych, alokowanie pamięci i wywoływanie innych metod, które muszą być wywołane, aby obiekt działał poprawnie.

Konstruktor

- Konstruktory mają tę samą nazwę jak klasa i nie mają typu zwracanego, a wywoływany jest on automatycznie, gdy utworzony zostanie nowy obiekt za pomocą operatora new. W C# możemy zdefiniować wiele konstruktorów dla jednej klasy, różniących się ilością i typem parametrów.
- Przykład na kolejnym slajdzie

Konstruktor

```
class Person
{
    public string name;
    public int age;

    // konstruktor bezparametrowy
    public Person()
    {
        name = "Unknown";
        age = 0;
    }

    // konstruktor z parametrami
    public Person(string name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
```

Konstruktor

- W tym przykładzie, klasa **Person** ma dwa konstruktory.

a) Pierwszy konstruktor jest bezparametrowy i ustawia domyślne wartości dla pól name i age.

b) Drugi konstruktor przyjmuje dwa parametry: **name** i **age** a następnie ustawia wartości **name** i **age** na te przekazane jako argumenty.

- Aby rozróżnić, która zmienna jest polem klasy, a która parametrem konstruktrora, używa się słowa **this**.

Konstruktor

- W języku C#, this jest słowem kluczowym, które odnosi się do bieżącego obiektu, na rzecz którego jest wywoływane, a zastosowanie go w konstruktorze pozwala na odwołanie się do właściwości lub metod bieżącego obiektu, które mają takie same nazwy jak parametry przekazywane do konstruktora. Na przykład, jeśli klasa ma pole o nazwie w, a konstruktor przyjmuje parametr o nazwie w, można użyć this.w w konstruktorze, aby odwołać się do pola w przykładzie z kolejnego slajdu:

Konstruktor

```
class MyClass
{
    private int w;
    public MyClass(int w)
    {
        this.w = w; // przypisanie wartości parametru do pola "w" bieżącego obiektu
    }
}
```

W tym przykładzie, gdy tworzymy nowy obiekt klasy **MyClass** z wywołaniem konstruktora, przekazując wartość **5** do parametru **w**, to pole w **w** bieżącym obiekcie zostanie ustawione na wartość **5.**

Destruktor

- Destruktor w języku C# jest specjalną metodą, która jest wywoływana automatycznie, kiedy obiekt danej klasy jest usuwany z pamięci przez Garbage Collector. Destruktor służy do zwalniania zasobów, które zostały zaalokowane w trakcie życia obiektu.
- W C# destruktor jest oznaczany znakiem tyldy (~) przed nazwą klasy i nie może on przyjmować argumentów.
- W przykładzie na kolejnym slajdzie zdefiniowano klasę MyClass, która posiada prywatne pole myResource, będące obiektem zaalokowanym w pamięci.

Destruktor

Destruktor, którego celem jest zwolnienie zasobów:

```
public class MyClass
{
   private object myResource;
   public MyClass()
   {
      myResource = new object();
   }
   ~MyClass()
   {
      // zwalnianie zasobów
      myResource = null;
   }
}
```

Destruktor

- W konstruktorze klasy MyClass inicjalizujemy pole myResource nowym obiektem. W destruktorze zwalniamy ten zasób poprzez ustawienie go na wartość null. W tym przypadku nie jest to konieczne, ponieważ Garbage Collector i tak zwolniłby ten obiekt, ale może to być przydatne w przypadku zasobów, które nie są zarządzane przez Garbage Collector, np. uchwyty do plików i połączeń sieciowych.

Destruktor

- Destruktor jest wywoływany automatycznie, gdy obiekt klasy jest usuwany z pamięci i nie możemy go wywołać ręcznie, tak jak konstruktora. W C# nie mamy też pewności, kiedy dokładnie zostanie wywołany destruktor, ponieważ zależy to od pracy Garbage Collector i wielu innych czynników. Destruktor w C# jest przydatnym narzędziem, ale powinien być używany z umiarem.
- Zbyt częste tworzenie destruktorów może prowadzić do spadku wydajności aplikacji, ponieważ Garbage Collector musi śledzić i zwalniać zasoby.

Destruktor

- W większości przypadków nie jest konieczne tworzenie destruktora, ponieważ Garbage Collector i tak zwolni zaalokowane zasoby, ale jeśli klasa używa zasobów, które nie są zarządzane przez Garbage Collector, takie jak na przykład wspomniane uchwyty do plików lub połączeń sieciowych, to destruktor może być przydatny do zwolnienia tych zasobów w kontrolowany i bezpieczny sposób.

Tworzenie obiektów

- Obiekt w języku C# jest instancją klasy i reprezentuje konkretny element lub zjawisko. Jest on tworzony na podstawie definicji klasy i zawiera zbiór zmiennych, które określają jego stan oraz zestaw metod, które umożliwiają wykonywanie na nim różnych operacji.
- Tworzenie obiektów w C# odbywa się poprzez wywołanie konstruktora klasy za pomocą operatora **new**.

Tworzenie obiektów

- Przykładowo, zdefiniujmy klasę **Car**, która będzie reprezentować **samochód**(klasa **Car** z **konstruktorem**):

```
public class Car
{
    public string make;
    public string model;
    public int year;
    public Car(string make, string model, int
        year)
    {
        this.make = make;
        this.model = model;
        this.year = year;
    }
}
```

Tworzenie obiektów

- W powyższym przykładzie klasa Car posiada trzy pola: make, model i year, które określają markę, model i rok produkcji samochodu. Konstruktor klasy Car przyjmuje trzy argumenty: make, model i year i inicjuje odpowiednie pola.

Tworzenie obiektów

- Możemy utworzyć obiekt klasy Car za pomocą następującego kodu:

```
Car myCar = new Car("Ford", "Mustang", 2020);
```

- W powyższym kodzie tworzymy nowy obiekt klasy Car za pomocą operatora new i przekazujemy mu trzy argumenty: "Ford", "Mustang" i 2020. Obiekt ten zostanie zainicjowany w taki sposób, że pole make będzie miało wartość "Ford", pole Model będzie miało wartość "Mustang" i pole Year będzie miało wartość 2020.

Tworzenie obiektów

- Możemy odwołać się do pól obiektu klasy Car za pomocą operatora kropki:

```
Console.WriteLine("My car is a {0} {1} built in {2}",
Car. Make, Car.Model, Car.Year);
```

- W powyższym kodzie wyświetlamy informacje o samochodzie, używając wartości pól obiektu myCar. Możemy tworzyć wiele obiektów tej samej klasy.
- Każdy z nich będzie miał swój własny stan, niezależny od innych obiektów. Istnieje możliwość wykorzystania tego, aby tworzyć bardziej skomplikowane programy, w których wiele obiektów klasy działa jednocześnie.

Politechnika Wrocławska

Dziękuje bardzo za uwagę

Dr inż. Paweł Maślak