Zawartość:

- interfejs polimorficzny

- interfejs

Co to jest interfejs?

W C# interfejs to konstrukcja, która przypomina klasy abstrakcyjne. Przypomina, ponieważ w klasie abstrakcyjnej możemy implementować, a w interfejsie nie. Interfejsy są w 100% abstrakcyjne (nie ma tam żadnych definicji) oraz zaczynają się od dużej litery I (i). Interfejsy to np. IObservable, IEnumerable, IEnumerator.

Przykład:

```
0 references
public interface ISortable
{
    0 references
    void setData(double[] data);
    0 references
    double[] getAscending();
    0 references
    double[] getDescending();
}
```

Pytanie po co nam coś takiego? Powody są dwa.

Pierwszy (ten prostszy) to fakt, że w C# można dziedziczyć tylko z jednej klasy. C# daje możliwość dziedziczenia z wielu interfejsów.

Drugi powód to fakt, że interfejsy zarówno jak klasy abstrakcyjne to świetne narzędzia do sporządzania umów. Wszystko co znajduje się w interfejsie musi zostać przynajmniej zdeklarowane w klasie dziedziczącej (w przypadku klas abstrakcyjnych także wszystkie abstrakcyjne metody muszą być przynajmniej zdeklarowane). Piszę „przynajmniej zdeklarowane”, bo można je zostawić puste (puste klamry).

O co chodzi z tą umową?

```
1 reference
public interface ISortable
{
    1 reference
    void setData(double[] data);
    1 reference
    double[] getAscending();
    0 references
    double[] getDescending();
}

0 references
internal class Program
{
    0 references
    static void SortDataAscending(ISortable sortable, double[] data)
    {
        sortable.setData(data);
        data = sortable.getAscending();
    }

    0 references
    static void Main(string[] args)
    {

    }
}
```

Metoda **SortDataAscending** przyjmuje w parametrze obiekt dowolnej klasy, która implementuje interfejs **ISortable**.

Tak więc raz możemy podać klasę SortBySelection : ISortable {…}, a raz BubbleSort : ISortable {…}

Dzięki interfejsowi gwarantujemy metodzie, że podany obiekt, będzie miał wskazane w interfejsie metody. Oczywiście mogą one być zaimplementowane inaczej.

Podobna sytuacja ma miejsce w przypadku kompozycji. Bardzo często tworzy się pola na obiekt typu interfejsu by potem za pomocą wstrzykiwania zależności podsunąć różne implementacje spełniające daną umowę (dany interfejs).

Interfejs polimorficzny – to określenie łączące interfejsy i klasy abstrakcyjne.

```
2 references
public interface ISortable
{
    2 references
    void setData(double[] data);
    1 reference
    double[] getAscending();
    0 references
    double[] getDescending();
}

0 references
public class BubbleSort : ISortable
{
    2 references
    public void setData(double[] data)
    {
        //some implementation
    }
}
```

Jak widać implementacja części interfejsu nie przejdzie.

W interfejsie podajemy tylko publiczne metody bo one będą pełnić rolę API danej klasy. W kontekście umowy nie interesują nas metody pomocnicze w danej klasie. Klasa dziedzicząca musi mieć dowolną ilość metod, ale te podane w interfejsie muszą być zaimplementowane (chociażby na pusto).