Właściwości i metody dostępowe, kolekcje

Właściwości – łączą w sobie cechy zwykłych zmiennych (pól) i metod. Właściwości mogą posiadać dwa rodzaje metod, zwanych metodami dostępowymi: `get` i `set`. Metoda `get` jest wywoływana, gdy klient odczytuje pewne dane. Metoda `set` jest wywoływana, gdy klient zapisuje dane. Klient nie wywołuje żadnej z tych metod explicite – wywoływane są one niejawnie. W poniższym przykładzie zdefiniowano właściwość o nazwie `Promien` i wyposażoną ją w obydwie metody dostępowe.

```
class Circle
{
      private double _promien;

      // definicja właściwości o nazwie Promien
      public double Promien
      {
            // metoda akcesorowa typu get
            // pozwala na odczytanie prywatnego pola _promien
            get
            {
                  return _promien;
            }

            // metoda akcesorowa typu set
            // pozwala na wprowadzenie nowych danych
            // do prywatnego pola _promien
            set
            {
            // przed wpisanieniem nowej wartosci
            // dokonywana jest prosta walidacja
            if (value > 0)
                  _promien = value;
            }
      }

      public Circle(double r)
      {
            _promien = r;
      }
}
```

Właściwość może zawierać obydwie metody dostępowe lub też tylko w jedną z nich. Dobrze zaprojektowana klasa powinna ukrywać swoją wewnętrzną strukturę i udostępniać jedynie pewien zbiór metod (interfejs). Właściwości i metody dostępowe wpisują się w ten postulat. Z punktu widzenia klienta, właściwości zachowują się tak jak publiczne pola klasy. Składnia polecenia do odczytania (lub zapisania) jest identyczna jak w przypadku zwykłego publicznego pola klasy. Jednak z punktu widzenia klasy zawierającej właściwości, takie operacje są de facto wywołaniem odpowiednich metod. Skoro są to metody, to można w nich wykonać pewne czynności. Stwarza to możliwość kontroli tego, co klient próbuje zrobić. W przypadku metody `set` najczęściej przeprowadza się walidację danych. W przypadku metody `get` może to być odczytanie danych, jeśli znajdują się w innym miejscu (np. w innym obiekcie lub w bazie danych) lub odpowiednie przygotowanie danych.

```
public static void CircleTest()
{
      // nowy obiekt klasy Circle
      Circle c = new Circle(2);

      // wypisanie promienia
      // "pracuje" metoda get
      Console.WriteLine(c.Promien); // metoda wypisze 2

      // wprowadzenie nowego (poprawnego) promienia
      // "pracuje" metoda set
      c.Promien = 3;

      // wypisanie promienia
      // "pracuje" metoda get
      Console.WriteLine(c.Promien);  // metoda wypisze 3

      // wprowadzenie nowego (tym razem niepoprawnego) promienia
      // wykorzystywana jest ponownie metoda set
      // ktora tym razem nie pozwoli na wprowdzenie wartosci -2
      c.Promien = -2;

      // wypisanie promienia
      // "pracuje" metoda get
      Console.WriteLine(c.Promien);   // metoda wypisze ponownie 3
}
```

Kolekcje – klasy, których podstawowym zadaniem jest przechowywanie innych obiektów. Przykładem kolekcji jest tablica. Może ona przechowywać dowolną liczbę obiektów, z tym że jej maksymalny rozmiar trzeba określić w momencie tworzenia i nie może on być zmieniany. Tego rodzaju ograniczenia dyskwalifikują tablice w niektórych zastosowaniach. C# oferuje kilka klas o cechach kolekcji, pozbawionych wad, jakie posiadają klasyczne tablice. Należą do nich m.in. klasa `ArrayList` i klasa `Hashtable`.

- **`ArrayList`** – klasa do przechowywania obiektów, której rozmiar jest dynamicznie zwiększany, jeśli zachodzi taka potrzeba. Klasa udostępnia szereg użytecznych właściwości i metod, m.in. metodę `Add`, która służy do dodawania obiektów do kolekcji. Dostęp do obiektów znajdujących się już w kolekcji jest możliwy poprzez indeks (tak jak w klasycznych tablicach). Obiekty umieszczone w `ArrayList` zachowują kolejność w jakiej zostały wstawione.

- **`Hashtable`** – tablica haszująca, pozwala na przechowywanie par: klucz – wartość. Przy czym kolejne klucze jak i odpowiadające im wartości mogą być obiektami różnych typów. Rozmiar tablicy haszującej jest dynamicznie zwiększany w miarę dodawania kolejnych elementów. Podobnie jak `ArrayList` udostępnia metodę `Add` do dodawania par klucz-wartość. Tablica haszująca charakteryzuje się bardzo szybkim dostępem do swoich elementów (efekt algorytmu haszujacego). Obiekty umieszczone w `Hashtable` nie zachowują kolejności wstawiania.

```
public static void TestArrayList()
{
	// utworzenie nowego obiektu ArrayList
	ArrayList lista = new ArrayList();

	// utworzenie nowego obiektu Circle i dodanie go do listy
	Circle c1 = new Circle(2);
	lista.Add(c1);

	// utworzenie nowego obiektu Circle i dodanie go do listy
	// bez tworzenia nowej zmiennej
	lista.Add(new Circle(5));

	// ... i jescze jeden obiekt
	lista.Add(new Circle(3));

	// przeglądanie zawartości listy
	for (int i = 0; i < lista.Count; i++)
	{
		// do obiektu z listy odwołuje sie tak jak do zwyklej tablicy,
		// rzutowanie jest konieczne
		Circle c = (Circle)lista[i];
		Console.WriteLine(c.Promien);
	}
}
```

```
public static void TestHashtable()
{
	// utworzenie nowego obiektu Hashtable
	Hashtable phoneBook = new Hashtable();

	// dodanie do tablicy haszującej trzech numerów telefonicznych
	// kluczem (indeksem) są imiona;
	long numerAli = 6578341;
	phoneBook.Add("ala", numerAli);
	phoneBook.Add("ola", (long)5532464);
	phoneBook.Add("ela", (long)5599874);

	// wypisanie numeru osoby o imieniu ala po uprzednim sprawdzeniu,
	// czy odpowiedni wpis istnieje
	if (phoneBook.ContainsKey("ala"))
	{
		// do obiektu z tablicy haszującej  odwołuje sie tak
		// jak do zwykłej tablicy, z tym że indeksem jest
		// klucz użyty przy dodawaniu obiektu do tablicy
		long numer = (long)phoneBook["ala"];
		Console.WriteLine(numer);
	}
	else
	{
		Console.WriteLine("Phone book does't contain such entry!!!");
	}
}
```

Zadanie 10.1 W klasie `Konto` dodać metody dostępowe `get` do następujących pól: `imie`, `nazwisko` i `numer`. Przetestować nowe metody (właściwości) tworząc nowe konto i wypisując imię, nazwisko i numer przy użyciu metody get. Spróbować wprowadzić nowe imię korzystając z metody `get`.

Zadanie 10.2 Klasa `Bank`

Krok 1 Utworzyć klasę `Bank` o następujących polach: `nazwa` (typ `string`), `tablicaKont` (10-cio elementowa tablica obiektów typu `Konto`)

Krok 2 Napisać konstruktor, który przyjmuje nazwę banku jako argument. Konstruktor winien utworzyć tablice kont.

Krok 3 Dla klasy Bank napisać następujące metody:

- `Konto ZalozKonto(string imie, string nazwisko)` – metoda tworzy obiekt klasy `Konto` i zapisuje go do tablicy kont. Metoda powinna zadbać, by nie przekroczyć rozmiaru tablicy kont.
- `Konto ZnajdzKonto(long numer)` – metoda znajduje w `tablicaKont` konto o podanym numerze. Jeśli takiego konta nie ma, wypisuje odpowiedni komunikat „Konto o numerze xxx nie istnieje”.

Krok 4 Do klasy Konto dodać następującą metodę:

- `void Przelej(Bank bank, long numer, decimal kwota)` – Metoda przyjmuje trzy argumenty: obiekt klasy `Bank`, numer konta na ktory należy dokonać przelewu i kwotę przelewu. Metoda powinna znaleść konto o podanym numerze i dokonać przelewu.

Krok 5 Przetestować klasy `Bank` i `Konto` w następujący sposób:

- utworzyć obiekt klasy `Bank` i dwa obiekty klasy `Konto`
- wpłacić na pierwsze konto kwotę 1000 zł, a na drugie 500 zł
- przelać z pierwszego konta na drugie kwotę 500 zł, korzystając z nowej metody `Przelej`

Po każdej operacji wypisać stan konta

Zadanie 10. 3 W klasie `Bank` zmienić typ pola `tablicaKont` na `ArrayList`. W konstruktorze utworzyć instancję klasy `ArrayList` i przypisać ją do pola `tablicaKont`. Przetestować wprowadzone zmiany tworząc dwa konta, wpłacając na nie dowolne kwoty i przelewając z jednego konta na drugie.

Zadanie 10.4. W klasie `Bank` zamienić typ pola `tablicaKont` na `HashTable`. Wprowadzić odpowiednie zmiany w metodzie `ZalozKonto`. Metoda po utworzeniu nowego konta powinna je zapisywać do tablicy haszującej w postaci pary: klucz – wartość. Kluczem będzie numer konta, a wartością obiekt klasy `Konto`. Taki sposób zapisu pozwala na szybsze szukanie konta o podanym numerze (wystarczy wykorzystać metodę `ContainsKey` z klasy `Hashtable`) W związku z powyższym wprowadzić odpowiednie zmiany w metodzie `ZnajdzKonto`.

Zadanie 10.5 Utworzyć klasy dla 3 różnych rodzajów pracowników: `PracownikGodzinowy`, `PracownikAkordowy` i `PracownikProwizyjny`. Wspólnym elementem wszystkich pracowników są pola: `imie` i `nazwisko`. Dla każdego z tych pól utworzyć metody dostępowe typu `get` i `set`. Każda z klas powinna również zawierać bezparametrową metodę `Wyplata`, zwracającą wartość typu `decimal`.

- Klasa `PracownikGodzinowy` zawiera następujące dodatkowe pola (i metody dostępowe `get` i `set`): `stawkaGodzinowa` (typ `decimal`), `liczbaGodzin` (typ `int`). Konstruktor powinien przyjmować 4 argumenty: `imie`, `nazwisko`, `stawkaGodzinowa` i `liczbaGodzin`. Wypłatę dla pracownika godzinowego oblicza się według następującego algorytmu: (i) jeśli liczba godzin jest mniejsza lub równa od 160, to wypłata wynosi stawka godzinowa pomnożona przez liczbę godzin, (ii) jeśli liczba godzin jest większa od 160, to za pierwsze 160 godzin liczone jest tak jak wyżej, a za nadgodziny stawka zwiększa się 1,5 razy.

- Klasa `PracownikAkordowy` zawiera następujące dodatkowe pola (i metody dostępowe `get` i `set`): `stawkaAkordowa` (typ `decimal`), `liczbaJednostek` (typ `int`). Konstruktor powinien przyjmować 4 argumenty: `imie`, `nazwisko`, `stawkaAkordowa` i `liczbaJednostek`. Wypłatę dla pracownika akordowego oblicza się w następujący sposób: stawka akordowa jest mnożona przez liczbę jednostek wytworzonego produktu.

- Klasa `PracownikProwizyjny` zawiera następujące dodatkowe pola (i metody dostępowe `get` i `set`): `pensjaPodstawowa` (typ `decimal`), prowizja (typ `decimal`) `liczbaJednostek` (typ `int`). Konstruktor powinien przyjmować 5 argumentów: `imie`, `nazwisko`, `pensjaPodstawowa`, `prowizja` i `liczbaJednostek`. Wypłatę dla pracownika akordowego oblicza się w następujący sposób: do pensji podstawowej dodawana jest prowizja pomnożona przez liczbę jednostek sprzedanego produktu.