Zajęcia 2. Dodanie do projektu klasy `Lista`

2.1. Wstęp

Poprawne tworzenie i inicjalizowanie obiektów typu **`Pracownik`** jest warunkiem koniecznym do zapewnienia pełnej funkcjonalności tworzonej aplikacji (zarówno w wersji pod konsolę jak i okienkowej). Kolejnym etapem budowy programu jest umożliwienie dokonywania operacji na pewnej liczbie pracowników ujętych w listę. Do takich operacji należeć powinny: dodawanie, usuwanie, wyszukiwanie pracowników jak również sortowanie, wczytywanie z klawiatury i wypisywanie na ekran ich danych. Aby to umożliwić wygodnie jest stworzyć klasę **`Lista`**, która udostępni odpowiednie metody służące do wykonywania wyżej wymienionych czynności na liście. Klasę tę należy dodać do projektu **AplikacjaPracownicy** w analogiczny sposób jak klasy zdefiniowane w poprzedniej instrukcji.

2.2. Klasa `Lista`

Aby zapewnić bezpieczeństwo typów i wyeliminować ewentualną konieczność rzutowania lub sprawdzania typów obiektów, klasa **`Lista`** musi posiadać pole składowe, które przechowuje elementy w formie kolekcji generycznej. Kolekcje generyczne zawarte są w przestrzeni **`System.Collections.Generic`** i udostępniają zbliżoną funkcjonalność co zwykłe kolekcje (przestrzeń **`System.Collections`**) – czyli kontenery przechowujące obiekty typu **`Object`** (np.: klasa **`ArrayList`**). Jak wspomniano we wstępie, tworzona aplikacja ma umożliwiać wykonywanie operacji na grupie pracowników (tj. na obiektach typu **`Pracownik`**), dlatego typ generyczny kolekcji należy zdefiniować jako **`<Pracownik>`**. W związku z tym w klasie **`Lista`** musi zostać zadeklarowane (prywatne) pole składowe typu **`List<T>`**, gdzie **`T`** oznacza typ **`Pracownik`**. Będzie ono służyć jako kontener do przechowywania obiektów typu **`Pracownik`** (a w przyszłości także typów pochodnych). Nazwa pola może być dowolna – w instrukcji przyjęto nazwę **`lista`**. Aby z kolei dokonywać wybranych operacji na liście, w klasie należy dodatkowo zdefiniować następujące publiczne metody:

- Konstruktor domyślny inicjalizujący pole składowe **`lista`** poprzez wywołanie konstruktora klasy **`List`**.
- **`void Dodaj(Pracownik pracownik)`** dodającą pracownika do listy pracowników.
- **`void WstawWPolozenie(int indeks, Pracownik pracownik)`** dodającą do listy pracownika przekazanego w argumencie metody w położenie **`indeks`**.
- **`int Usun(string nazwisko)`** usuwającą z listy pracownika o zadanym w argumencie nazwisku i zwracającą indeks usuniętego pracownika. Jeżeli na liście nie znajduje się pracownik o danym nazwisku, metoda ma zwrócić **`-1`**.
- **`void Usun(int indeks)`** usuwającą pracownika z listy znajdującego się na pozycji **`indeks`**. W przypadku niepoprawnej wartości argumentu metody (np. wartość mniejsza od **`0`**), na ekranie pojawić się ma stosowny komunikat.
- **`Pracownik Szukaj(string nazwisko)`** wyszukującą na liście pracownika o zadanym w argumencie nazwisku. Jeżeli na liście znajduje się taki pracownik, metoda ma zwrócić znaleziony obiekt. W przeciwnym wypadku metoda ma zwrócić **`null`**.
- **`void Sortuj()`** sortującą elementy listy.
- **`void ZapisConsole()`** wypisującą na ekran wszystkich pracowników z listy.
- **`void OdczytConsole()`** wczytującą z klawiatury dane do nowostworzonego obiektu klasy **`Pracownik`** i dodającą obiekt do listy.
- **`void Wyczysc()`** czyszczącą listę, tj.: usuwającą z listy wszystkie obiekty typu **`Pracownik`**.

W klasie należy również zdefiniować dwie właściwości:

- **`public Pracownik this[int i]`** – indekser zwracający **`i`**-ty element listy.
- **`public int Rozmiar`** zwracającą liczbę elementów listy.

2.3. Klasa `List<T>` i jej wybrane metody

Jak już wspomniano w punkcie 2.2, klasa **`List<T>`** jest generycznym odpowiednikiem klasy **`ArrayList`**, czyli kolekcji typu tablica. Przez twórców platformy .NET została zaprojektowana tak, iż jej rozmiar jest automatycznie zwiększany w miarę potrzeb. Informację na temat liczby przechowywanych elementów

w kolekcji można uzyskać poprzez odwołanie się do właściwości **Capacity**. Podczas inicjalizacji obiektu typu **List<T>**, właściwość ta przyjmuje wartość **0**, a po dodaniu pojedynczego elementu zwiększana jest do **4**. Po każdym zapełnieniu kolekcji zwiększana jest o **4**.
Klasa **List<T>** definiuje ponadto szereg właściwości i metod, które mogą zostać wykorzystywane do przeszukiwania i modyfikowania zawartości kolekcji. Operacje na tablicy obejmują takie działania jak wstawianie i usuwanie w wybranych miejscach, dodawanie, czyszczenie, poszukiwanie określonych obiektów na liście. W przeciwieństwie do kolekcji typu **ArrayList**, klasa udostępnia również mechanizm obsługi wyrażeń lambda (**=>**). Dzięki temu w bardzo prosty sposób można na przykład zaimplementować operacje przeszukiwania tablicy. Poniżej przedstawiono kilka wybranych składowych kolekcji **List<T>**, które warto wykorzystać podczas definiowania metod oraz właściwości klasy **Lista**:

- **void Add(T t)**: metoda służąca do dodawania obiektu **t** na koniec listy (dla typów referencyjnych **t** może przyjmować wartość **null**).
- **void Insert(int indeks, T t)**: metoda służąca do wstawiania obiektu **t** we wskazane przez argument **indeks** miejsce (numeracja elementów kolekcji rozpoczyna się od indeksu **0**).
- **bool Remove(T t)**: metoda służąca do usuwania obiektu **t** z listy. Warto zwrócić uwagę, że metoda zwraca wartość typu **bool**: **true** w przypadku, kiedy obiekt **t** został poprawnie usunięty z listy, **false** wówczas, gdy obiekt nie został znaleziony w kolekcji.
- **void RemoveAt(int indeks)**: metoda służąca do usuwania obiektu ze wskazanego przez argument **indeks** miejsca.
- **void Sort()**: metoda służąca do sortowania elementów listy. Za pomocą właściwości domyślnego komparatora **Comparer<T>.Default** dla typu **T** ustala kolejność elementów. W tym celu sprawdzane jest, czy typ **T** implementuje interfejs **IComparable<T>**. Jeżeli tak, do celów sortowania używana jest implementacja interfejsu. W przeciwnym wypadku właściwość **Comparer<T>.Default** sprawdza czy typ **T** implementuje interfejs **IComparable**. Jeżeli kompilator wykryje brak odpowiedniej implementacji, rzucany jest wyjątek **InvalidOperationException**.
- **void Clear()**: metoda służąca do usuwania wszystkich elementów listy.
- **bool Contains(T t)**: metoda służąca do sprawdzenia czy w kolekcji znajduje się obiekt **t** przekazany w argumencie. Jeżeli taki obiekt istnieje – zwracana jest wartość **true**, w przeciwnym wypadku – **false**.

Wskazówka: Aby zdefiniować wybrane metody dla klasy **Lista**, warto skorzystać z wyżej opisanych metod zaimplementowanych w klasie **List<T>**. Można je wywoływać na rzecz obiektu tej klasy.

2.4. Definicja wybranych metod klasy Lista

Dodawanie pracownika do listy

Argumentem metody **Dodaj**, którą należy zdefiniować dla klasy **Lista** jest obiekt typu **Pracownik**. Na pierwszy rzut oka wydawałoby się zatem, że wywołanie w tej metodzie metody **Add** (opisanej w podrozdziale 2.3) klasy **List<T>** i przesyłanie jej jako argument tego właśnie obiektu wystarczy do dodania elementu do listy. Takie rozwiązanie jest poprawne wyłącznie w przypadku, gdy metoda **Dodaj** wywołana jest w sposób przedstawiony na Listingu 2.1.

```
Lista l = new Lista();
l.Dodaj(new Pracownik("Jan","Kowalski",1,"lutego" 1999,"Mickiewicza","1","Krakow"));
l.Dodaj(new Pracownik("Piotr","Nowak",2,"maja",2001,"Sienkiewicza","2","Rzeszow"));
```

Listing 2.1: Dodawanie do listy obiektów tworzonych przy użyciu konstruktora klasy **Pracownik**.

Jej argumentem jest wówczas instancja nowego obiektu stworzonego za pomocą operatora **new**. Dodawanie pracownika do listy nie musi jednak polegać na wielokrotnym wywołaniu konstruktora klasy **Pracownik** przy każdym wywołaniu metody **Dodaj**. Do listy obiekty można przecież dodawać w prostszy sposób (bez konieczności tworzenia nowego egzemplarza klasy za każdym razem), pozwalając dodatkowo użytkownikowi

wpisywać dowolne dane z klawiatury. Kod źródłowy przedstawiony na Listingu 2.2 obrazuje bardziej elastyczne rozwiązanie.

```
Pracownik p = new Pracownik();
p.OdczytConsole();
l.Dodaj(p);
p.OdczytConsole();
l.Dodaj(p);
```

Listing 2.2: Wielokrotne dodawanie do listy jednego obiektu typu **Pracownik** o zmodyfikowanych składowych.

W tym wypadku obiekt klasy **Pracownik** tworzony jest wyłącznie raz (wywołanie konstruktora domyślnego) i po wprowadzeniu odpowiedniej informacji z klawiatury poprzez wywołanie metody **OdczytConsole**, dodawany jest do listy. Poprawne działanie powyższego kodu (mimo, że nie ma tutaj błędów zarówno składniowych jak i logicznych) jest uzależnione od definicji metody **Dodaj** klasy **Lista**. Jeżeli metoda zdefiniowana jest w sposób pokazany na Listingu 2.3,

```
public void Dodaj(Pracownik pracownik)
{
   lista.Add(pracownik);
}
```

Listing 2.3: Niepoprawna definicja metody **Dodaj**.

to przy każdym wywołaniu metody **OdczytConsole** na rzecz obiektu **p**, dojdzie do modyfikacji zawartości całej kolekcji: wszystkie elementy przyjmą wartości ostatnio modyfikowanego obiektu **p**. Stanie się tak dlatego, że metoda **Add** pracuje na referencji dodawanego obiektu, czyli na jego oryginale. Wpisanie nowych danych dla pracownika w programie i dodanie go do listy pociąga za sobą modyfikację całej grupy pracowników – jest to zjawisko bardzo niebezpieczne, gdyż nie jest kontrolowane przez użytkownika. Zatem jak zdefiniować metodę **Dodaj**, aby kolejne wywołania metody **OdczytConsole** nie niszczyły zgromadzonych do tej pory danych?

Odpowiedź na to pytanie można znaleźć w różnicy wywołań metody **Add** na rzecz obiektu **lista** oraz metody **Dodaj** na rzecz obiektu **l**. Warto zwrócić uwagę na to, że jeżeli argumentem przesyłanym do metody **Dodaj** jest obiekt klasy **Pracownik** tworzony za pomocą operatora **new**, to wówczas metoda **Add** dodaje nowo stworzoną instancję do kolekcji typu **List<Pracownik>** i nie dochodzi do jakiejkolwiek modyfikacji elementów listy. Natomiast w przypadku przesyłania raz stworzonego obiektu, który ulega zmianom poprzez interfejs z użytkownikiem, metoda **Add** dodaje ten sam obiekt a elementy kolekcji stają się identyczne. Rozwiązanie tego problemu tkwi w argumencie przesyłanym do metody **Add**. Metodę tę należy wywołać w taki sposób, by po każdym wywołaniu metody **Dodaj** po modyfikacji obiektu **p**, do listy dodawać nowy obiekt, a nie ten sam. Ale skąd przy każdym wywołaniu metody **Dodaj** brać nowy obiekt typu **Pracownik**? Najprostszym rozwiązaniem jest przesłanie do metody **Add** jako argument instancji typu **Pracownik** stworzonej na wzór argumentu metody **Dodaj**. Będzie to nic innego jak wywołanie konstruktora kopiującego tej klasy (rozdział 1.6). Metoda **Dodaj** przyjmie postać taką jak na Listingu 2.4.

```
public void Dodaj(Pracownik pracownik)
{
   lista.Add(new Pracownik(pracownik));
}
```

Listing 2.4: Poprawna definicja metody **Dodaj**.

Rozwiązanie zilustrowane na Listingu 2.4 zapewni poprawne dodawanie pracowników do listy bez jakichkolwiek niebezpiecznych (i niekontrolowanych) modyfikacji jej elementów i na tym etapie działania i funkcjonalności aplikacji jest jak najbardziej wystarczające. Teraz metodę **Dodaj** można wywoływać na obydwa sposoby: przesyłając jej tworzony obiekt (wywołania konstruktora) lub obiekt już istniejący, do którego użytkownik wielokrotnie podaje jakieś dane.

Należy jednak już w tym momencie podkreślić, że zaprezentowane podejście nie jest uniwersalne z punktu widzenia idei programowania obiektowego. Ze względu na to, że klasa **Pracownik** będzie stanowić typ podstawowy dla nowych, bardziej wyspecjalizowanych klas, metodę **Dodaj** dobrze jest już teraz ulepszyć do postaci zaprezentowanej na Listingu 2.5.

```
public void Dodaj(Pracownik pracownik)
{
   lista.Add(pracownik.Clone());
}
```

Listing 2.5: Optymalna definicja metody **Dodaj** umożliwiająca zastosowanie polimorfizmu przy dodawaniu obiektów do listy.

Jak można zauważyć różnica w definicji metody **Dodaj** sprowadza się jedynie do tego, że zamiast wywołania konstruktora kopiującego klasy **Pracownik**, do metdody **Add** przysyłany jest rezultat zwracany przez metodę **Clone** wywołaną na rzecz obiektu **pracownik**. Metoda **Clone** jest publiczną wirtualną metodą składową klasy **Pracownik**, która ma zwracać instancję nowostworzonego obiektu na wzór referencji **this**. Jest więc ona odpowiednikiem konstruktora kopiującego. Na chwilę obecną wprowadzona modyfikacja daje ten sam rezultat, ale na przyszłość – wprowadza polimorfizm i upraszcza strukturę programu.

Wskazówka: W podobny sposób należy zdefiniować metodę **WstawWPolozenie**, która dodaje do listy pracownika w wybrane miejsce.

Wyszukiwanie pracownika na liście

Znalezienie na liście pracownika o zadanym nazwisku sprowadza się do przeszukania zawartości całej kolekcji i sprawdzenia czy nazwisko kolejnego obiektu równe jest nazwisku przekazanemu do metody jako argument. Można w tym celu użyć pętli **foreach** lub **for** (**while**, **do while**). W momencie gdy pracownik o zadanym nazwisku zostanie znaleziony, metoda ma zwrócić jego instancję. W przypadku przejścia przez całą listę i nie znalezienia obiektu, wartością zwracaną metody ma być referencja **null**. Listing 2.6 przedstawia definicję metody **Szukaj**.

```
public Pracownik Szukaj(string nazwisko)
{
   foreach (Pracownik p in lista)
   {
      if (p.Nazwisko.Equals(nazwisko))
      {
         return p;
      }
   }
   return null;
}
```

Listing 2.6: Definicja metody **Szukaj** do znajdywania pracownika na liście o zadanym nazwisku.

2.5. Metoda Main

W metodzie głównej **Main** klasy **Program** należy zdefiniować obiekt klasy **Lista** i wywołać wybrane metody na rzecz tego obiektu, które umożliwią dodawanie, usuwanie, wyszukiwanie, sortowanie pracowników na liście.