ВБУДОВАНІ ІНТЕРФЕЙСИ .NET

Питання 5.3.

Інтерфейси IEnumerable та IEnumerator

- foreach-конструкцію можна застосовувати до будь-якого типу, який підтримує метод GetEnumerator().
  - Повернемось до класів Car і Radio.
  - Класс Garage зберігає набір об'єктів Car (автомобіль) всередині System.Array.
  - Було б зручно виконувати обхід елементів об'єкта Garage, використовуючи конструкцію foreach.
  - Компілятор повідомить, що в класі Garage не реалізовано метод GetEnumerator(), який формально визначено в інтерфейсі lEnumerable з простору імен System.Collections.

```
// Garage contains a set of Car objects.
public class Garage
{
  private Car[] carArray = new Car[4];

  // Fill with some Car objects upon startup.
  public Garage()
  {
    carArray[0] = new Car("Rusty", 30);
    carArray[1] = new Car("Clunker", 55);
    carArray[2] = new Car("Zippy", 30);
    carArray[3] = new Car("Fred", 30);
  }
}
```

```
// Это выглядит корректным...
public class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with IEnumerable / IEnumerator *****\n");
    Garage carLot = new Garage();
    // Проход по всем объектам Car в коллекции?
    foreach (Car c in carLot)
    {
      Console.WriteLine("{0} is going {1} MPH",
        c.PetName, c.CurrentSpeed);
    }
    Console.ReadLine();
  }
}
```

Інтерфейси IEnumerable та IEnumerator

- Визначення стандартного інтерфейсу IEnumerable:

```
// Этот интерфейс информирует вызывающий код о том,
// что подэлементы объекта могут перечисляться.
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

- Метод GetEnumerator() повертає посилання на інтерфейс System.Collections.IEnumerator.
  - Цей інтерфейс надає інфраструктуру, яка дозволяє викликаючому коду здійснювати обхід внутрішніх об'єктів, що містяться в lEnumerable-сумісному контейнері.

```
// Этот интерфейс позволяет вызывающему коду получать подэлементы контейнера.
public interface IEnumerator
{
  bool MoveNext();        // Переместить вперед внутреннюю позицию курсора.
  object Current { get; } // Получить текущий элемент (свойство,
                          //   доступное только для чтения).
  void Reset();           // Поместить курсор перед первым элементом.
}
```

Інтерфейси IEnumerable та IEnumerator

```
using System.Collections;
...
public class Garage : IEnumerable
{
  // System.Array уже реализует IEnumerator!
  private Car[] carArray = new Car[4];

  public Garage()
  {
    carArray[0] = new Car("FeeFee", 200);
    carArray[1] = new Car("Clunker", 90);
    carArray[2] = new Car("Zippy", 30);
    carArray[3] = new Car("Fred", 30);
  }

  public IEnumerator GetEnumerator()
  {
    // Возвратить IEnumerator объекта массива
    return carArray.GetEnumerator();
  }
}
```

- При модифікації типу Garage для підтримки цих інтерфейсів можна піти довгим шляхом і реалізувати кожен метод вручну.
  - Простіший шлях: делегувати запит до System.Array, оскільки цей тип уже реалізує інтерфейси IEnumerable та IEnumerator.
- Змінивши так тип Garage, можна безпечно використовувати його всередині конструкції foreach.
  - Враховуючи, що метод GetEnumerator() був визначений як відкритий, користувач об'єкта може також взаємодіяти з IEnumerator.
  - Якщо потрібно приховати функціональність IEnumerable на рівні об'єкта, досить скористатися явною реалізацією інтерфейсу:

```
IEnumerator IEnumerable.GetEnumerator()
{
  // Возвратить IEnumerator объекта массива
  return carArray.GetEnumerator();
}
```

```
// Работать напрямую с IEnumerator.
IEnumerator i = carLot.GetEnumerator();
i.MoveNext();
Car myCar = (Car)i.Current;
Console.WriteLine("{0} is going {1} MPH", myCar.PetName, myCar.CurrentSpeed);
```

Побудова методів ітератора із застосуванням ключового слова yield

```
public class Garage : IEnumerator
{
  private Car[] carArray = new Car[4];
  ...
  // Метод итератора.
  public IEnumerator GetEnumerator()
  {
    foreach (Car c in carArray)
    {
      yield return c;
    }
  }
}
```

- Ітератор – це член, який вказує, як повинні повертатися внутрішні елементи контейнера при обробці в циклі foreach.
  - Хоч метод ітератора, як і раніше, повинен іменуватися GetEnumerator(), а його вихідне значення мати тип IEnumerator, створюваний спеціальний клас не потребує реалізації будь-яких очікуваних інтерфейсів.
  - У цій реалізації методу GetEnumerator() прохід по піделементах здійснюється з використанням внутрішньої логіки foreach, а кожен об'єкт Car повертається викликаючому коду із застосуванням синтаксису yield return.
  - Ключове слово yield слугує для вказівки значення (або значень), яке повинно повертатися конструкції foreach у викликаючому коді.

- Методи ітераторів не зобов’язані використовувати ключеве слово foreach для повернення свого вмісту.
  - Допускається також таке визначення методу ітератора:

```
public IEnumerator GetEnumerator()
{
  yield return carArray[0];
  yield return carArray[1];
  yield return carArray[2];
  yield return carArray[3];
}
```

Побудова іменованого ітератора

```
public IEnumerable GetTheCars(bool ReturnRevesed)
{
  // Возвратить элементы в обратном порядке.
  if (ReturnRevesed)
  {
    for (int i = carArray.Length; i != 0; i--)
    {
      yield return carArray[i-1];
    }
  }
  else
  {
    // Возвратить элементы в том порядке, в каком они размещены
    foreach (Car c in carArray)
    {
      yield return c;
    }
  }
}
```

Ключове слово yield формально може використовуватись всередині будь-якого методу, незалежно від його назви.

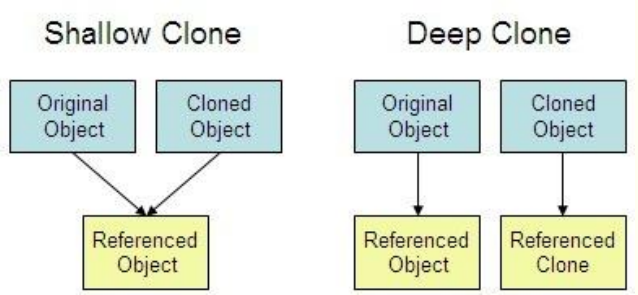
- Такі методи (***іменовані ітератори***) унікальні в тому, що можуть приймати довільну кількість аргументів.
- При побудові іменованого ітератора дуже важливо розуміти, що метод буде повертати інтерфейс lEnumerable, а не очікуваний IEnumerator-сумісний тип.
- Цей новий метод дозволяє викликаючому коду отримувати піделементи як у прямому, так і в зворотному порядку, якщо у вхідному параметрі передається значення true.

Тепер з ним можна взаємодіяти

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with the Yield Keyword *****\n");
  Garage carLot = new Garage();

  // Получить элементы, используя GetEnumerator().
  foreach (Car c in carLot)
  {
    Console.WriteLine("{0} is going {1} MPH",
      c.PetName, c.CurrentSpeed);
  }

  Console.WriteLine();

  // Получить элементы (в обратном порядке!), используя именованный итератор.
  foreach (Car c in carLot.GetTheCars(true))
  {
    Console.WriteLine("{0} is going {1} MPH",
      c.PetName, c.CurrentSpeed);
  }
  Console.ReadLine();
}
```

- іменовані ітератори – корисні конструкції, тому що дозволяють визначати в єдиному спеціальному контейнері кілька способів, запитування вихідного набору.

Інтерфейс ICloneable




```
// Класс по имени Point.
public class Point
{
  public int X {get; set;}
  public int Y {get; set;}

  public Point(int xPos, int yPos) { X = xPos; Y = yPos;}
  public Point(){}

  // Переопределить Object.ToString().
  public override string ToString()
  {  return string.Format("X = {0}; Y = {1}", X, Y ); }
}
```

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Cloning *****\n");

  // Две ссылки на один и тот же объект!
  Point p1 = new Point(50, 50);
  Point p2 = p1;
  p2.X = 0;
  Console.WriteLine(p1);
  Console.WriteLine(p2);
  Console.ReadLine();
}
```

- У System.Object визначено метод MemberwiseClone().
  - Він використовується для отримання поверхневої (неглибокої) копії поточного об'єкта.
  - Користувачі об'єкта не можуть викликати цей метод безпосередньо, тому що він є захищеним, проте сам об'єкт може це робити в ході клонування.
- Для прикладу визначимо клас Point, що представляє точку.
  - При присвоєнні одній змінній посилального типу іншої отримуємо два посилання, що вказують на один і той же об'єкт у пам'яті.
  - Модифікація з використанням будь-якоuj з цих посилань впливає на той же самий об'єкт у кучі.
  - Щоб забезпечити спеціальний тип здатністю повертати викликаючому коду ідентичну копію самого себе, можна реалізувати стандартний інтерфейс ICloneable.

Інтерфейс ICloneable

- ICloneable визначає єдиний метод Clone():

```
public interface ICloneable
{
  object Clone();
}
```

  - Очевидно, что реалізація методу Clone () в різних класах варіюється.
  - Проте базова функціональність зазвичай незмінна: копіювання значень змінних-членів у новий об'єкт того ж типу та повернення його користувачеві.

```
// Теперь Point поддерживает возможность клонирования.
public class Point : ICloneable
{
  public int X { get; set; }
  public int Y { get; set; }

  public Point(int xPos, int yPos) { X = xPos; Y = yPos; }
  public Point() { }

  // Переопределить Object.ToString().
  public override string ToString()
  { return string.Format("X = {0}; Y = {1}", X, Y); }

  // Возвратить копию текущего объекта.
  public object Clone()
  { return new Point(this.X, this.Y); }
}
```

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Cloning *****\n");
  // Обратите внимание, что Clone() возвращает простой тип object.
  // Для получения нужно производного типа требуется явное приведение
  Point p3 = new Point(100, 100);
  Point p4 = (Point)p3.Clone();

  // Изменить p4.X (что не приводит к изменению p3.x).
  p4.X = 0;

  // Вывести все объекты.
  Console.WriteLine(p3);
  Console.WriteLine(p4);
  Console.ReadLine();
}
```

Інтерфейс ICloneable

- Поточну реалізацію Point можна дещо покращити.
  - Оскільки Point не містить внутрішніх змінних посилального типу, реалізація методу Clone() спрощується:

```
public object Clone()
{
  // Копировать каждое поле Point почленно.
  return this.MemberwiseClone();
}
```

- Якби в Point містились внутрішні змінні посилального типу, метод MemberwiseClone() копіював би посилання на ці об'єкти (створював поверхневу копію).
  - Для підтримки глибокої (детальної) копії в ході клонування потрібно створити новий екземпляр кожної змінної посилального типу.

Складніший приклад клонування

- Нехай клас Point містить змінну-член посилального типу PointDescription.
  - Підтримується назва точки та її ідентифікаційний номер (System.Guid - глобально унікальний ідентифікатор, globally unique identifier, GUID) — статистично унікальне 128-бітне число.
  - метод Clone () поки не модифіковано.
  - Початкові зміни класу Point включають модифікацію методу ToString() для врахування нових даних стану, а також визначення та створення посилального типу PointDescription.

```
public class PointDescription
{
  public string PetName {get; set;}
  public Guid PointID {get; set;}

  public PointDescription()
  {
    PetName = "No-name";
    PointID = Guid.NewGuid();
  }
}
```

```
public class Point : ICloneable
{
  public int X { get; set; }
  public int Y { get; set; }
  public PointDescription desc = new PointDescription();

  public Point(int xPos, int yPos, string petName)
  {
    X = xPos; Y = yPos;
    desc.PetName = petName;
  }
  public Point(int xPos, int yPos)
  {
    X = xPos; Y = yPos;
  }
  public Point() { }

  // Переопределить Object.ToString().
  public override string ToString()
  {
    return string.Format("X = {0}; Y = {1}; Name = {2};\nID = {3}\n",
    X, Y, desc.PetName, desc.PointID);
  }

  // Возвратить копию текущего объекта.
  public object Clone()
  { return this.MemberwiseClone(); }
}
```

Складніший приклад клонування

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Cloning *****\n");
  Console.WriteLine("Cloned p3 and stored new Point in p4");
  Point p3 = new Point(100, 100, "Jane");
  Point p4 = (Point)p3.Clone();

  Console.WriteLine("Before modification:");
  Console.WriteLine("p3: {0}", p3);
  Console.WriteLine("p4: {0}", p4);
  p4.desc.PetName = "My new Point";
  p4.X = 9;

  Console.WriteLine("\nChanged p4.desc.petName and p4.X");
  Console.WriteLine("After modification:");
  Console.WriteLine("p3: {0}", p3);
  Console.WriteLine("p4: {0}", p4);
  Console.ReadLine();
}
```

```
***** Fun with Object Cloning *****

Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

p4: X = 100; Y = 100; Name = Jane;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509


Changed p4.desc.petName and p4.X
After modification:
p3: X = 100; Y = 100; Name = My new Point;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509

p4: X = 9; Y = 100; Name = My new Point;
ID = 133d66a7-0837-4bd7-95c6-b22ab0434509
```

Складніший приклад клонування

- Щоб змусити метод Clone() створювати повну детальну копію внутрішніх посилальних типів, потрібно сконфігурувати об'єкт, що повертає метод MemberwiseClone() для врахування імені поточного об'єкта Point (тип System.Guid насправді є структурою, тому числові дані будуть дійсно копіюватись).
  - Об'єкт Point, що повертає метод Clone() дійсно копіює свої внутрішні змінні-члени посилального типу:

```
public object Clone()
{
  // Сначала получить поверхностную копию.
  Point newPoint = (Point)this.MemberwiseClone();

  // Теперь заполнить пробелы.
  PointDescription currentDesc = new PointDescription();
  currentDesc.PetName = this.desc.PetName;
  newPoint.desc = currentDesc;
  return newPoint;
}
```

```
***** Fun with Object Cloning *****

Cloned p3 and stored new Point in p4
Before modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 100; Y = 100; Name = Jane;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a


Changed p4.desc.petName and p4.X
After modification:
p3: X = 100; Y = 100; Name = Jane;
ID = 51f64f25-4b0e-47ac-ba35-37d263496406

p4: X = 9; Y = 100; Name = My new Point;
ID = 0d3776b3-b159-490d-b022-7f3f60788e8a
```

Інтерфейс IComparable

- Інтерфейс System.IComparable описує поведінку, яка дозволяє сортувати об'єкт на основі зазначеного ключа.

```
// Этот интерфейс позволяет объекту указать
// его отношение с другими подобными объектами.
public interface IComparable
{
  int CompareTo(object o);
}
```

  - просто додано нову властивість для представлення унікального ідентифікатора кожного автомобіля і модифікований конструктор

```
public class Car
{
...
   public int CarID {get; set;}
   public Car(string name, int currSp, int id)
   {
     CurrentSpeed = currSp;
     PetName = name;
     CarID = id;
   }
   ...
}
```

Нехай існує масив об’єктів Car

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Sorting *****\n");
  // Создать массив объектов Car.
  Car[] myAutos = new Car[5];
  myAutos[0] = new Car("Rusty", 80, 1);
  myAutos[1] = new Car("Mary", 40, 234);
  myAutos[2] = new Car("Viper", 40, 34);
  myAutos[3] = new Car("Mel", 40, 4);
  myAutos[4] = new Car("Chucky", 40, 5);
  Console.ReadLine();
}
```

- Клас System.Array визначає статичний метод Sort().
  - При його виклику над масивом внутрішніх типів (int, short, string і т.д.) є можливість сортувати елементи масиву в числовому або алфавітному порядку.
- Що станеться при передачі методу Sort() масиву об’єктів Car?
  - **`// Чи відсортуються об’єкти Car? Поки що ні!`**
    `Array.Sort(myAutos);`
  - Запуск тестового коду призведе до винятку, тому що клас Car не підтримує потрібний інтерфейс.

```
// Итерация по объектам Car может быть упорядочена на основе CarID.
public class Car : IComparable
{
  ...
  // Реализация IComparable.
  int IComparable.CompareTo(object obj)
  {
    Car temp = obj as Car;
    if (temp != null)
    {
      if (this.CarID > temp.CarID)
        return 1;
      if (this.CarID < temp.CarID)
        return -1;
      else
        return 0;
    }
    else
      // Параметр не является объектом типа Car!
      throw new ArgumentException("Parameter is not a Car!");
  }
}
```

- При побудові спеціальних типів для забезпечення можливості сортування масивів, які містять елементи цих типів, можна реалізувати інтерфейс IComparable.
  - Реалізуючи деталі CompareTo(), ви повинні самостійно прийняти рішення про те, що буде критерієм впорядкування.
  - Для типу Car цілком логічним кандидатом є внутрішня властивість CarID

Вихідні значення ConpareTo()

- Вихідне значення CompareTo() використовується для з'ясування того, чи є поточний об'єкт меншим, більшим чи рівним об'єкту, з яким він порівнюється:

Вихідне значення CompareTo()	Опис
Будь-яке число, менше за нуль	Даний екземпляр знаходиться перед вказаним об'єктом у порядку сортування
Нуль	Даний екземпляр дорівнює вказаному об'єкту
Будь-яке число, більше за нуль	Даний екземпляр знаходиться після вказаного об'єкта в порядку сортування

- Попередню реалізацію CompareTo() можна вдосконалити, враховуючи, що в C# тип даних int (System.Int32 в CLR) реалізує інтерфейс IComparable.

```
int IComparable.CompareTo(object obj)
{
  Car temp = obj as Car;
  if (temp != null)
    return this.CarID.CompareTo(temp.CarID);
  else
    // Параметр не является объектом типа Car!
    throw new ArgumentException("Parameter is not a Car!");
}
```

Використання інтерфейсу IComparable

```
// Использование интерфейса IComparable.
static void Main(string[] args)
{
  // Создать массив объектов Car.
  ...
  // Отобразить содержимое текущего массива.
  Console.WriteLine("Here is the unordered set of cars:");
  foreach(Car c in myAutos)
    Console.WriteLine("{0} {1}", c.CarID, c.PetName);

  // Теперь отсортировать массив, используя IComparable!
  Array.Sort(myAutos);
  Console.WriteLine();

  // Отобразить содержимое отсортированного массива.
  Console.WriteLine("Here is the ordered set of cars:");
  foreach(Car c in myAutos)
    Console.WriteLine("{0} {1}", c.CarID, c.PetName);
  Console.ReadLine();
}
```

```
***** Fun with Object Sorting *****

Here is the unordered set of cars:
1 Rusty
234 Mary
34 Viper
4 Mel
5 Chucky

Here is the ordered set of cars:
1 Rusty
4 Mel
5 Chucky
34 Viper
234 Mary
```

Вказівка кількох порядків сортування за допомогою IComparer

- У даній версії класу Car в якості критерію сортування використовується ідентифікатор автомобіля (carID).
  - В іншому проектному рішенні для цього могло б застосовуватися ім'я автомобіля (для перерахування автомобілів в алфавітному порядку).
  - А що якщо потрібно побудувати клас Car, який підтримував би сортування за обома критеріями?
  - Для цього повинен використовуватися інший стандартний інтерфейс – IComparer, визначений в просторі імен System.Collections:

```
interface IComparer
{
  int Compare(object o1, object o2);
}
```

- На відміну від IComparable, інтерфейс IComparer зазвичай не реалізується в типі, що підлягає сортуванню (тобто Car).
  - Замість цього даний інтерфейс реалізується в будь-якій кількості допоміжних класів, по одному для кожного порядку сортування (по дружньому імені, ідентифікатору автомобіля і т.д.).
  - Щоб дозволити користувачеві об'єкта сортувати масив об'єктів Car по дружньому імені, слід створити додатковий допоміжний клас, який реалізує IComparer.

Вказівка кількох порядків сортування за допомогою IComparer

- Тепер цей допоміжний клас можна використовувати в коді.
  - У System.Array є кілька перевантажених версій методу Sort(), одна з яких бере об'єкт, який реалізує інтерфейс IComparer.

```
// Этот вспомогательный класс используется для сортировки
// массива объектов Car по дружественному имени.
public class PetNameComparer : IComparer
{
  // Проверить дружественное имя каждого объекта.
  int IComparer.Compare(object o1, object o2)
  {
    Car t1 = o1 as Car;
    Car t2 = o2 as Car;
    if(t1 != null && t2 != null)
      return String.Compare(t1.PetName, t2.PetName);
    else
      throw new ArgumentException("Parameter is not a Car!");
  }
}
```

```
static void Main(string[] args)
{
  ...
  // Теперь сортировать по дружественному имени.
  Array.Sort(myAutos, new PetNameComparer());

  // Вывести отсортированный массив.
  Console.WriteLine("Ordering by pet name:");
  foreach(Car c in myAutos)
    Console.WriteLine("{0} {1}", c.CarID, c.PetName);
  ...
}
```

Спеціальні властивості та спеціальні типи сортування

```
// Теперь поддерживается специальное свойство
// для возврата корректного интерфейса IComparer.

public class Car : IComparable
{
  ...
  // Свойство, возвращающее PetNameComparer.
  public static IComparer SortByPetName
  { get { return (IComparer)new PetNameComparer(); } }
}
```

- Можна використовувати спеціальну статичну властивість для надання користувачу об'єкта допомоги з сортуванням типів Car за специфічним елементом даних.
  - Нехай у клас Car додано статичну, доступну тільки для читання властивість SortByPetName, яка повертає екземпляр об'єкта, що реалізує інтерфейс IComparer (тут – PetNameComparer).
- У коді користувача об’єкта тепер можна сортувати за назвою з використанням жорстко асоційованої властивості, а не автономного класу PetNameComparer:
  - **`// Сортування за назвою тепер дещо простіша.`**
    `Array.Sort(myAutos, Car.SortByPetName);`
  - Слід зазначити, що інтерфейси зустрічаються в кожному ключовому просторі імен .NET, тому в майбутньому доведеться мати справу з різноманітними стандартними інтерфейсами.

ДЯКУЮ ЗА УВАГУ!

Наступне питання: Параметричний поліморфізм. Узагальнені типи даних