Задание №6.
Стандартные интерфейсы

Цели задания:

- Изучить механизм полиморфизма объектно-ориентированного программирования с использованием стандартных интерфейсов .NET.

В прошлом задании мы разработали интерфейсы и создали полиморфные классы. Полиморфизм является одним из трёх ключевых принципов ООП наравне с инкапсуляцией и наследованием, который позволяет существенно повысить гибкость архитектуры приложений, их масштабируемость. Идея использования полиморфизма заложена и в платформу .NET. Работа некоторых стандартных классов основана на работе через интерфейсы и полиморфизм.

В данном задании мы попробуем реализовать и использовать стандартные интерфейсы .NET:

- Интерфейс ICloneable как альтернатива конструкторам копирования;
- Интерфейс IEquatable и переопределение операции сравнения двух объектов класса;
- Интерфейс IComparable и переопределение операции сравнения двух объектов класса.

ICloneable

До появления языка C#, в языке программирования Си++ наравне с обычными конструкторами существовали (и существуют) так называемые конструкторы копирования. Конструктор копирования - это конструктор класса, в качестве параметра которого передается объект этого же класса. Задача конструктора копирования - создать объект, являющийся копией исходного объекта. Как правило, это означало копирование значений всех полей другого объекта. Пример:

```
public class Person
{
    public string Name { get; set; }
```

```
    public long Phone { get; set; }

    // Обычный конструктор класса с параметрами
    public Person(string name, long phone)
    {
        Name = name;
        Phone = phone;
    }

    // Конструктор копирования
    public Person(Person person)
    {
        Name = person.Name;
        Phone = person.Phone;
    }
}

public static class Demo
{
    public static void ShowCopyingConstructor()
    {
        // Сначала создаём обычный объект
        var person = new Person("Юрий Смирнов", 79991112233);

        // Если нам нужно создать копию объекта,
        // мы можем воспользоваться конструктором копирования
        var copy = new Person(person);
    }
}
```

В C# также можно создавать конструкторы копирования, но как альтернатива им появился интерфейс для создания копий объектов ICloneable:

```
namespace System;

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

Создавать собственный интерфейс ICloneable не нужен, он уже создан и находится в пространстве имен System. Вам как разработчику необходимо добавить

реализацию этого интерфейса в свои классы. Реализация интерфейса ICloneable для класса Person:

```
public class Person : ICloneable
{
    public string Name { get; set; }

    public long Phone { get; set; }

    // Обычный конструктор класса с параметрами
    public Person(string name, long phone)
    {
        Name = name;
        Phone = phone;
    }

    // Реализация ICloneable вместо конструктора копирования
    public object Clone()
    {
        return new Person(this.Name, this.Phone);
    }
}
```

Вызов метода Clone() будет выглядеть следующим образом:

```
public static class Demo: ICloneable
{
    public static void ShowCopyingConstructor()
    {
        // Сначала создаём обычный объект
        var person = new Person("Юрий Смирнов", 79991112233);

        // Если нам нужно создать копию объекта,
        // мы можем воспользоваться методом Clone()
        var copy = (Person)person.Clone();
    }
}
```

Как видно, по сложности реализации оба подхода (с интерфейсом ICloneable и конструктором копирования) не отличаются и практически идентичны.

Преимуществом интерфейса ICloneable и его же недостатком является универсальность подхода. Конструкторы копирования, как и любые конструкторы, вызываются с явным указанием имени класса, в то время как копирования объекта в интерфейса ICloneable осуществляется через единый метод Clone().

С одной стороны, так как все копируемые объекты имеют общий интерфейс и общий метод Clone(), мы можем сделать единую обработку для копирования объектов. Например, если мы разрабатываем графический редактор, и нам нужно сделать копию выделенных пользователем примитивов – квадратов, прямоугольников, кругов, линий и пр. То есть множества объектов ***разных*** типов данных. Нам достаточно поместить их в коллекцию List<ICloneable> и вызывать у каждого объекта метод клонирования.

```
var selectedObjects = List<ICloneable>();
…
var copiedObjects = List<object>();
foreach (var selectedObject in selectedObjects)
{
   copiedObjects.Add(selectedObject.Clone());
}
```

С конструкторами копирования такой приём уже не удастся, так как для объекта каждого типа надо явно вызывать конструктор копирования по имени этого класса.

С другой стороны, чтобы сделать интерфейс ICloneable универсальным, разработчикам .NET пришлось сделать возвращаемый тип данных метода Clone() тип object – базовый тип данных для всех типов данных в .NET. Это значит, что при создании копии одного объекта нам требуется сделать явное преобразование к целевому типу данных:

```
Person copy = (Person)person.Clone();
```

То есть, если мы хотим скопировать объекты и сохранить исходный тип данных, а не положить объекты в обобщенную коллекцию, то реализация ICloneable не даёт преимуществ, а вызов обычного конструктора копирования в таком случае выглядит синтаксически проще.

Можно подытожить, что подход с интерфейсом ICloneable является более универсальным – благодаря интерфейсу можно создавать копии целых коллекций объектов. При этом, если в исходном коде нужно выполнить разовые копии отдельных объектов, подход также годится. Если же вам приходится часто делать

копии объектов какого-то типа, и вы хотите немного улучшить читаемость кода, можете реализовать конструктор копирования. Реализовывать же оба подхода в одном классе, как правило, не целесообразно – разработчик должен делать классы лаконичными, а написание двух открытых методов, которые решают одну и ту же задачу, избыточно.

Стоит также сказать, что сами стратегии копирования объектов могут быть разные. Если ваш класс состоит из простых типов данных, то достаточно скопировать значения этих полей. Но если ваш класс, например, агрегирует другие сложные объекты, это может создать дилемму. На что стоит обращать внимание при реализации копирования объектов:

- Что делать с полями типа Id? В копии объекта должен быть тот же Id или новый?
- Что делать с полями, которые хранят время создания объекта? В некоторых задачах бывает необходимо новый объект отметить временным штампом. Объект-копия может быть создан значительно позже исходного объекта. Мы должны скопировать исходное время создания объекта или сохранить время создания объекта-копии?
- Если копируемый объект (например, Customer) агрегирует другой объект (Address), мы должны: 1) в копию объекта скопировать ссылку на объект первоначального адреса или; 2) в копию объекта должны сделать копию агрегируемого объекта? В одном случае редактирование объекта адреса приведет к изменениям в двух покупателях. Во втором случае, при необходимости мы сможем редактировать адреса и покупателей независимо друг от друга, а также могут быть накладные расходы ресурсов на копирование внутренних объектов.
- Если копируемый объект (например, Order) агрегирует коллекцию других объектов (List<Item>) мы должны: 1) скопировать ссылку на исходный List<Item>; 2) мы должны создать новый List<Item>, но скопировать в него объекты из исходного списка; 3) мы должны создать новый List<Item> и скопировать в него копии исходных объектов? Каждый из трёх вариантов даёт принципиально разное поведение при дальнейшем редактировании наших объектов Order.
- Должны ли мы в принципе копировать все поля объекта? Или есть данные, копирование которых необязательно?
- Если сравнить исходный и скопированный объект через оператор сравнения `==` или метод `Equals()`, какой результат мы должны получить?

Однозначного ответа ни на один из перечисленных вопросов нет. Всё зависит от предметной области, копируемых классов, а также цели копирования. В одной и той же программе может понадобится одновременно несколько стратегий копирования объектов одного класса.

IEquatable

Каждый тип данных в C# наследует от базового класса `object` метод `bool Equals(object other)`. Метод используется для сравнения двух объектов на равенство наравне с оператором `==`. Операция сравнения двух объектов является фундаментальной как в программировании, так и в логике, так и в математике. Без операции сравнения невозможно, например, реализовать поиск, фильтрацию или категоризацию данных – базовые операции по обработке информации.

Если мы хотим сравнить простой тип данных, такой как int, нам достаточно выполнить побитовое сравнение двух переменных в оперативной памяти. Отличие хотя бы в одном бите означает, что числа не равны. Но для составных объектов всё может быть сложнее. Аналогично задаче копирования объектов, возникают вопросы о сравнении объектов только по Id, необходимости сравнения временных штампов или сравнения агрегируемых коллекций. Стандартная реализация Equals() предполагает сравнение объектов по ссылке. Если в двух переменных хранится ссылка на один и тот же объект в памяти – то переменные равны. Если же переменные ссылаются на разные объекты, то они не равны, даже если значения всех их полей абсолютно идентично.

Стандартный метод Equals() является виртуальным, т.е. разработчик может переопределить его реализацию, если его интересует сравнение не просто по ссылке объектов:

```
public override bool Equals(object other)
{
   // Здесь пишется реализация сравнения объекта this и other
}
```

Как правило, при переопределении Equals() в начале надо убедиться, что входной объект не null, затем проверить его тип данных, затем сделать явное преобразование к собственному типу данных и сравнить объекты по ссылке:

```
public class Person
```

```
{
    public string Name { get; set; }

    public long Phone { get; set; }

...
    // Реализация ICloneable вместо конструктора копирования
    public override bool Equals(object other)
    {
        // Обязательные проверки прежде чем мы сравним поля
        if (other == null)
           return false;

        if (other is not Person)
           return false;

        if (object.ReferenceEquals(this, other))
           return true;

        var person2 = (Person)other;

        // Только теперь мы можем сделать собственное сравнение
        return (this.Name == person2.Name);
    }
}
```

В данной реализации мы считаем два объекта Person равными, если у них одинаковые имена, при этом номер телефона роли не играет. Мы можем сделать сравнение объектов по имени и телефону, можем сделать сравнение по имени, но без учета регистра и т.п. Реализация будет зависеть от задач. Обратите внимание, что возможность сравнивать объекты по ссылке у нас остается, так как мы всегда может вызвать стандартный метод сравнения ссылок объектов:

```
bool object.ReferenceEquals(object obj1, object obj2)
```

Кроме стандартного Equals(object), в C# есть интерфейс IEquatable, который добавляем в класс метод Equals(T), где T – это тип данных вашего класса, т.е. метод сравнения, но в качестве аргумента будет использоваться не обобщенная переменная object, а конкретный тип данных, например Person. Такой метод может упростить сравнение объектов одного типа и даже повысить производительность как самого сравнения, так и, например, поиска в стандартных коллекциях.

Еще одна причина для перегрузки метода Equals и реализации стандартного интерфейса IEquatable – это использование метода Equals при написании юнит-тестов. В юнит-тестах часто приходится сравнивать результат работы метода с некоторых эталонным объектом. Сравнение по ссылке здесь не подходит, и зачастую требуется сравнение по значениям полей. Чтобы упростить сравнение объектов в юнит-тестах, стоит реализовать интерфейс IEquatable и перегрузить методы сравнения.

Подробнее можно ознакомиться по ссылкам [1, 2, 3, 4].

IComparable и IComparable<T>

Стандартный интерфейс аналогичен интерфейсу IEquatable, предназначен для сравнения объектов между собой и определяет метод int CompareTo(T), где T – тип данных вашего класса. Метод должен возвращать 0, если объекты равны; 1, если исходный объект больше передаваемого в метод; и -1, если исходный объект меньше передаваемого в метод. Метод CompareTo() используется для операций сравнения `<`, `>`, `<=`, `>=`. В отличие от Equals(), CompareTo() не просто говорит о равенстве двух объектов, но явно указывает, какой из них больше, а какой меньше. Сравнение объектов важно для реализации сортировок, категоризаций и разного рода упорядочивания данных. Подробнее можно ознакомиться по ссылке [5].

Реализация стандартных интерфейсов

1. Создать из ветки develop ветку Tasks/6_standard_interfaces, перейти в новую ветку.
2. Данное задание будет одним из самых простых, так как не требует создания новых классов или изменений в пользовательском интерфейсе.
3. Реализуйте интерфейс ICloneable в классах Item, Address, Cart.
4. Реализуйте интерфейс IEquatable<> и перегрузку метода Equals() в классах Item, Address, Order.
5. Реализуйте интерфейс IComparable<> в классах Item (по стоимости), PointsDiscount (по баллам), PercentDiscount (по проценту скидки).
6. Проверьте правильность работы новых методов. Для тестирования можете создать отдельную вкладку в приложении, вёрстка на ваше усмотрение.
7. Проверьте правильность оформления кода, наличие комментариев. Для упрощения написания xml-комментариев можно использовать можно использовать тэг <inheritdoc />. Сделайте коммит.

Проектная документация

1. Нарисуйте новую диаграмму классов. На ней необходимо обозначить стандартные интерфейсы и связанные с ними классы. Стандартные интерфейсы, как правило, на указываются на диаграммах классов. Однако, в рамках задания стандартные интерфейсы должны быть отрисованы, чтобы закрепить обозначение реализации интерфейсов на диаграммах.
2. Обратите внимание, что новые методы добавились в ранее написанные классы – изменения должны быть показаны на диаграмме.
3. Диаграмму сохраните под названием «Practics6.*» в папке doc. Сохраните диаграмму как в формате программы, которую вы будете использовать, так и в формате png или jpeg, чтобы диаграмму можно было просмотреть онлайн через GitHub.
4. Выполнить сливание текущей ветки с веткой develop.

Дополнительные задания

1. Реализуйте ICloneable и IEquatable для классов Customer и Store. Сложность задания заключается в том, что Customer агрегирует сразу несколько коллекций, например, Orders, заказы внутри которого в свою очередь агрегируют

товары. Тщательно продумайте, как именно должно проходить клонирование таких сложных составных объектов.